Scrum-prosessimallin käyttöliittymäriskien minimointi simulointipohjaisella GDD-käyttöliittymäsuunnittelumenetelmällä

Mikko Romppainen

Helsinki 6.5.2010
Pro gradu -tutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta/Osasto – Fakultet/Sektion – Faculty/Section	Laitos – Institution – Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä – Författare – Author

Mikko Romppainen

Työn nimi – Arbetets titel – Title

Scrum-prosessimallin käyttöliittymäriskien minimointi simulointipohjaisella GDD-käyttöliittymäsuunnittelumenetelmällä

Oppiaine – Läroämne – Subject

Tietojenkäsittelytiede

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro gradu -tutkielma	6.5.2010	82 sivua

Tiivistelmä – Referat – Abstract

Scrum-prosessimalli jättää menettelyt ohjelmiston käyttöliittymän tuottamiseen auki ja käyttöliittymä syntyy pahimmillaan toimintolistan pohjalta ohjelmoinnin sivutuotteena. Näin syntynyt järjestelmä soveltuu suurella riskillä huonosti käyttötarkoitukseensa.

Tässä tutkielmassa analysoitiin Scrum-prosessimallin käyttöliittymäriskikohtia, joita löytyi kolme: asiakkaan toivomat ominaisuudet päätyvät sellaisinaan ohjelmiston vaatimuksiksi, toimintolistan pohjalta syntyy käytettävyydeltään heikkoja käyttöliittymäratkaisuja ja käyttöliittymän arviointi sprintin katselmoinnissa tuottaa epäluotettavia tuloksia.

Tutkielmassa käsitellään Scrum-prosessimallin käyttöliittymäriskien minimointia simulointipohjaisella GDD-käyttöliittymäsuunnittelulla. Riskien minimointia tarkastellaan esimerkkitapauksen avulla, jossa ohjelmistoyritys Reaktor toteutti ammattikorkeakoulun toiminnansuunnittelujärjestelmän vuosisuunnitteluosion. Esimerkkitapauksessa Scrumin käyttöliittymäriskit saatiin minimoitua selvittämällä loppukäyttäjien käyttötilanteet kontekstuaalisilla haastatteluilla, suunnittelemalla käyttöliittymä GDD-menetelmällä ja arvioimalla käyttöliittymää hyödyllisyysläpikäynneillä.

Alkuperäisessä Scrumissa liiketoiminnallisesta kannattavuudesta vastaava tuotteen omistaja ja toteutustiimi ottavat vastuulleen myös käyttöliittymän toimintalogiikan. GDD:n myötä vastuu toimintalogiikasta siirretään käyttöliittymäsuunnittelijalle, jolloin Scrumin roolit muuttuvat. Tässä työssä käsitellään GDD-käyttöliittymäsuunnittelun tuomat muutokset Scrumin rooleihin ja käytäntöihin.

Scrumin käyttöliittymäriskien minimoinnin jälkeen toteutusvaiheeseen jää vielä Scrumista riippumattomia käyttöliittymäriskejä. Tämän työn esimerkkitapauksessa keskeisin näistä oli käyttöliittymätoteutukseen päätyneet puutteelliset interaktiot. Riski eliminoitiin hyväksymismenettelyllä, jossa ohjelmoija antaa toteutetun ominaisuuden käyttöliittymäsuunnittelijalle tarkistettavaksi. Hyväksymismenettelyn avulla projektin työnjako selkiytyi, toteutustyön laatu parani ja toteutustiimin ja käyttöliittymäsuunnittelijoiden välinen kommunikaatio tehostui.

ACM Computing Classification System (CCS 1998):
D.2.1 Requirements/Specifications
H.5.2 User Interfaces
K.6.3 Software Management

Avainsanat – Nyckelord – Keywords

Scrum, ketterä ohjelmistokehitys, käyttöliittymät, simulointipohjainen käyttöliittymäsuunnittelu, GUIDe, GDD, vaatimusmäärittely, riskienhallinta, kenttätutkimus

Säilytyspaikka – Förvaringställe – Where deposited

Kumpulan tiedekirjasto, sarjanumero C-

Muita tietoja – Övriga uppgifter – Additional information

Sisältö

1 Johdanto

Ohjelmiston käyttöliittymän, eli toimintojen, tietosisällön ja interaktiotapojen tuottaminen on jätetty Scrum-menetelmässä [Schwaber01] auki. Tyypillisesti asiakas esittää vaatimuksiaan järjestelmän ominaisuuksista ja käyttöliittymä syntyy pahimmassa tapauksessa ohjelmoinnin sivutuotteena kokonaan ilman käyttöliittymäsuunnittelua. Tällä tavalla syntyneessä järjestelmässä on suuri riski, että se ei sovellu hyvin käyttötarkoitukseensa. Tässä työssä analysoidaan Scrum-prosessimallin käyttöliittymäriskikohtia. Tällä tarkoitetaan sellaisia prosessimallin menettelyjä, joiden seurauksena loppukäyttäjälle saattaa päätyä työtehtävien kannalta ongelmallinen käyttöliittymä.

Riskikohtien paikantamisen jälkeen tutkielmassa käsitellään Scrumin käyttöliittymäriskien minimointia GUIDe-prosessimallin simulointipohjaisella GDD-käyttöliittymäsuunnittelulla [Laakso04a, Laakso06b]. GUIDe-prosessimallissa loppukäyttäjille eteen tulevat käyttötilanteet selvitetään käyttäjätarkkailuilla ja kontekstuaalisilla haastatteluilla, jonka jälkeen käyttöliittymä suunnitellaan GDD-menetelmällä simuloimalla selvitettyjä tilanteita askel askeleelta. GDD tuottaa käyttöliittymän näyttökuvasarjat, joissa järjestelmän toiminnot, tietosisältö ja interaktiotavat on kiinnitetty. Käyttöliittymää suunnitellessa loppukäyttäjien työtehtävät simuloidaan paperiprototyyppien avulla läpi useaan kertaan ja paperiprototyypit testataan loppukäyttäjillä. Tällä tavalla saadaan varmistettua järjestelmän soveltuvuus käyttötarkoitukseensa.

Scrumin käyttöliittymäriskien minimointia GDD-suunnittelulla tarkastellaan ohjelmistoyritys Reaktorin esimerkkiprojektin avulla. Esimerkkitapauksessa Reaktor toteutti ammattikorkeakoulujen toiminnansuunnittelujärjestelmän vuosisuunnitteluosion GDD-suunnittelua ja Scrumia käyttäen. Scrum-menetelmän riskien lisäksi käsitellään projektin aikana eteen tulleet muut käyttöliittymäriskit. Lopuksi tarkastellaan, miten GDD-käyttöliittymäsuunnittelu vaikuttaa Scrum-menetelmään: miten Scrumin menettelyt ja roolit muuttuvat, ja millainen rooli käyttöliittymäsuunnittelijalla on Scrumin käytännöissä.

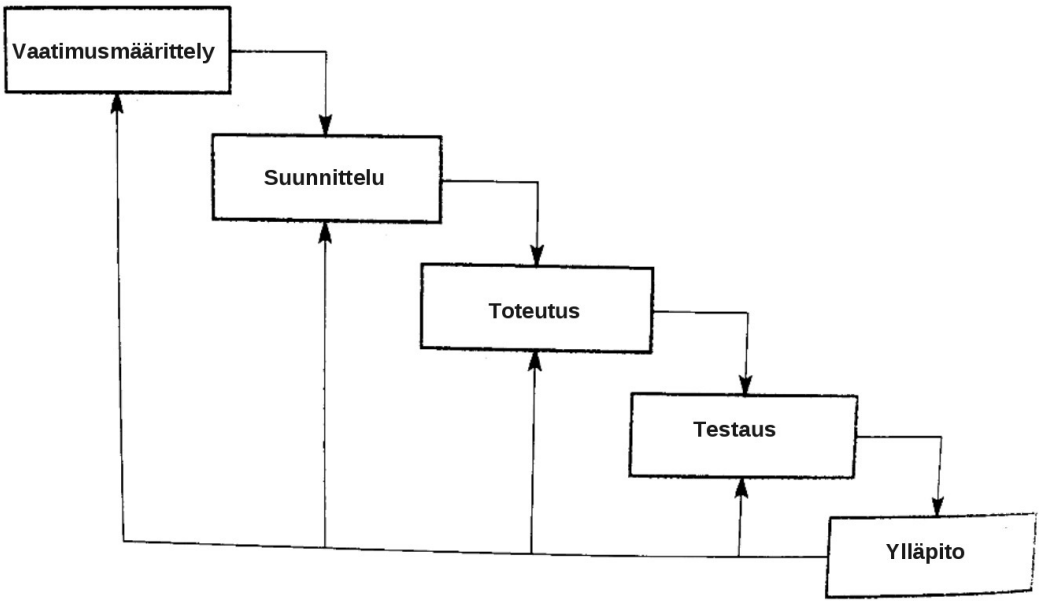
Luvussa 2 esitellään lyhyt historia ja motivaatio ketterille menetelmille. Projektinhallinnallisen kehyksen antavan Scrum-menetelmän lisäksi tarkemmin käsitellään Extreme Programming -prosessimalli (XP) [Beck99a], jonka hyväksi havaittuja ohjelmointikäytäntöjä käytetään usein yhdessä Scrum-menetelmän kanssa. Luvussa 3 esitellään GUIDen ja GDD:n vaiheet. Luvussa 4 analysoidaan Scrum-prosessimallista aiheutuvia käyttöliittymäriskejä. Luvussa 5 käsitellään Reaktorin esimerkkitapaus ja Scrumin käyttöliittymäriskien minimointi GDD-käyttöliittymä-suunnittelulla, sekä GDD:n vaikutukset Scrumin rooleihin ja käytäntöihin.

2 Ketterät prosessimallit

Tässä luvussa esitellään Scrum ja Extreme Programming (XP) -prosessimallit. Scrum tarjoaa projektinhallinnallisen kehyksen, jota täydennetään usein XP:n ohjelmointikäytännöillä. Luvussa 2.1 tehdään lyhyt katsaus prosessimallien historiaan ja tarkastellaan, miten ketterät menetelmät pyrkivät ratkaisemaan perinteisten prosessimallien ongelmia. Luvussa 2.2 tutustutaan Scrum-prosessimalliin. Extreme Programming esitellään luvussa 2.3 ja luvussa 2.4 katsotaan, miten XP:n ohjelmointikäytännöt toimivat yhdessä Scrum-menetelmän kanssa.

2.1 Vesiputousmallista ketteriin prosesseihin

Tiukasti vaiheittain etenevät ja yksityiskohtaisten dokumenttien tuottamiseen nojaavat prosessimallit ovat olleet käytössä ohjelmistotuotannossa jo vuosikymmeniä. Perinteisen vesiputousmallin esitteli ensimmäisen kerran Royce artikkelissaan vuonna 1970 [Royce70]. Vesiputousmallissa määrittely, suunnittelu, toteutus, testaus ja ylläpito on kukin erotettu omaksi vaiheekseen (kuva 2.1) [Sommerville07, s. 65-68]. Jokaisen vaiheen lopputuotos on dokumentaatio, joka toimii syötteenä seuraavalle vaiheelle. Edellinen vaihe suoritetaan loppuun ennen seuraavan aloittamista.



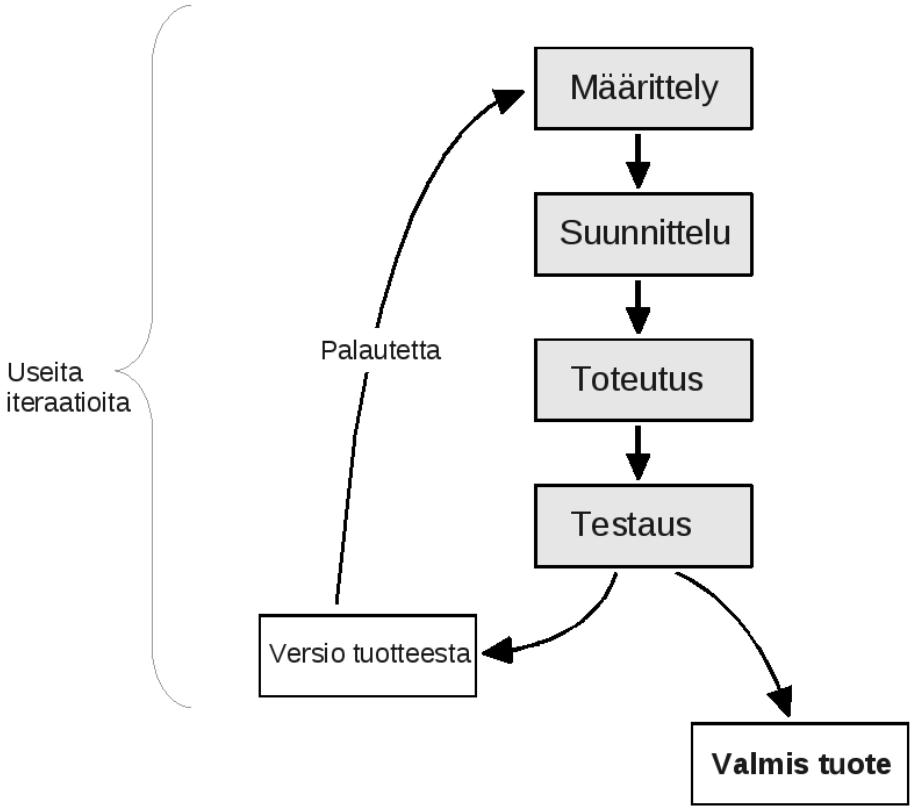
Kuva 2.1: Vesiputousmalli [Sommerville07, s. 66].

Vaikka Royce esitteli vesiputousmallin artikkelissaan ongelmallisena ja kehotti tekemään ensin nopean pilottiversion järjestelmästä puutteiden löytämiseksi, vakiintui kuvan 2.1 kaltainen malli pitkäksi aikaa ohjelmistotuotannon yleisimmin käytetyksi malliksi. Vesiputousmalli on edelleen laajasti käytössä: vuonna 2002 tehdyssä tutkimuksessa 35 prosenttia vastaajista ilmoitti käyttävänsä vesiputousmallia [Neill03].

Vesiputousmallin ongelmallisuus on kuitenkin tiedostettu jo pitkään. Malli pakottaa projektiryhmän kirjoittamaan yksityiskohtaiset dokumentit myös sellaisista kohdista, joita ei ymmärretä hyvin [Boehm88]. Tämä aiheuttaa paljon turhaa määrittely-, suunnittelu- ja ohjelmointityötä, kun ongelmalliset ratkaisut joudutaan tekemään alusta asti uudestaan. Vesiputousmallissa tekeillä oleva ohjelmisto konkretisoituu asiakkaalle vasta toteutusvaiheessa, jolloin muutosten aiheuttamat kustannukset saattavat olla kymmeniä kertoja suuremmat kuin kustannukset muutoksen teolle määrittelyvaiheessa [Boehm81]. Vuosituhannen vaihteessa tehdyssä laajassa tutkimuksessa selvisi, että projektien alkuvaiheissa dokumentoiduista ja lopulta toteutetuista toiminnallisuuksista 45 % oli sellaisia, joita kukaan ei koskaan käyttänyt [Larman07, s. 73].

Vesiputousmallin ongelmia pyrittiin välttämään toteuttamalla järjestelmä iteratiivisesti väliversioiden avulla. Tämä uusi idea ohjelmistokehitykselle esiteltiin kirjallisuudessa jo 1970-luvulla [mm. Gilb77 s. 214-217, Mills76] ja keskustelu vilkastui 1980-luvulla [mm. Boehm84, Gilb85, McCracken81]. Vuonna 1982 tehdyssä kokeessa seitsemän eri projektiryhmää toteutti oman versionsa samasta ohjelmistosta [Boehm84]: Neljä ryhmää käytti vaiheistettua prosessimallia ja tuotti yksityiskohtaiset dokumentit kaikista vaiheista. Kolme ryhmää toteutti ohjelmiston iteratiivisesti prototyyppien avulla. Prototyyppejä käytettäessä suurin piirtein saman lopputuloksen saavuttamiseen tarvittiin keskimäärin 45 % vähemmän työtä verrattuna dokumentteihin nojaavaan malliin.

Iteratiivinen ja inkrementaalinen ohjelmistokehitys saavutti suuren suosion kuitenkin vasta 1990-luvulla, jolloin useita evoluutiomallia (kuva 2.2) mukailevia prosessimalleja esiteltiin [Larman03]. Evoluutiomallissa ohjelmistosta toteutetaan



Kuva 2.2: Evoluutiomalli (lähteiden [Larman07 s. 10-11] ja [Sommerville07 s.68-69] pohjalta editoitu).

useita väliversioita [Sommerville07, s. 68-69]. Versiot voivat olla joko toimivia osia lopullisesta ohjelmistosta tai prototyyppejä, joiden avulla saadaan tarkennettua huonosti ymmärrettyjä kohtia vaatimuksissa. Näin tarvittavat muutokset voidaan tehdä aikaisemmassa vaiheessa projektia kuin vesiputousmallissa, jolloin niiden aiheuttamat kustannukset pysyvät paremmin kurissa. Vuosituhannen vaihteessa tehdyssä tutkimuksessa todettiin, että evoluutiomallilla saavutetaan laadukkaampi lopputulos ja nopeampi kehitysprosessi kuin vesiputousmallilla [MacCormack01]. Iteratiiviset ja inkrementaaliset menetelmät pelastivatkin useita vesiputousmallia käyttäviä projekteja, jotka olivat lähellä epäonnistumista [Larman03].

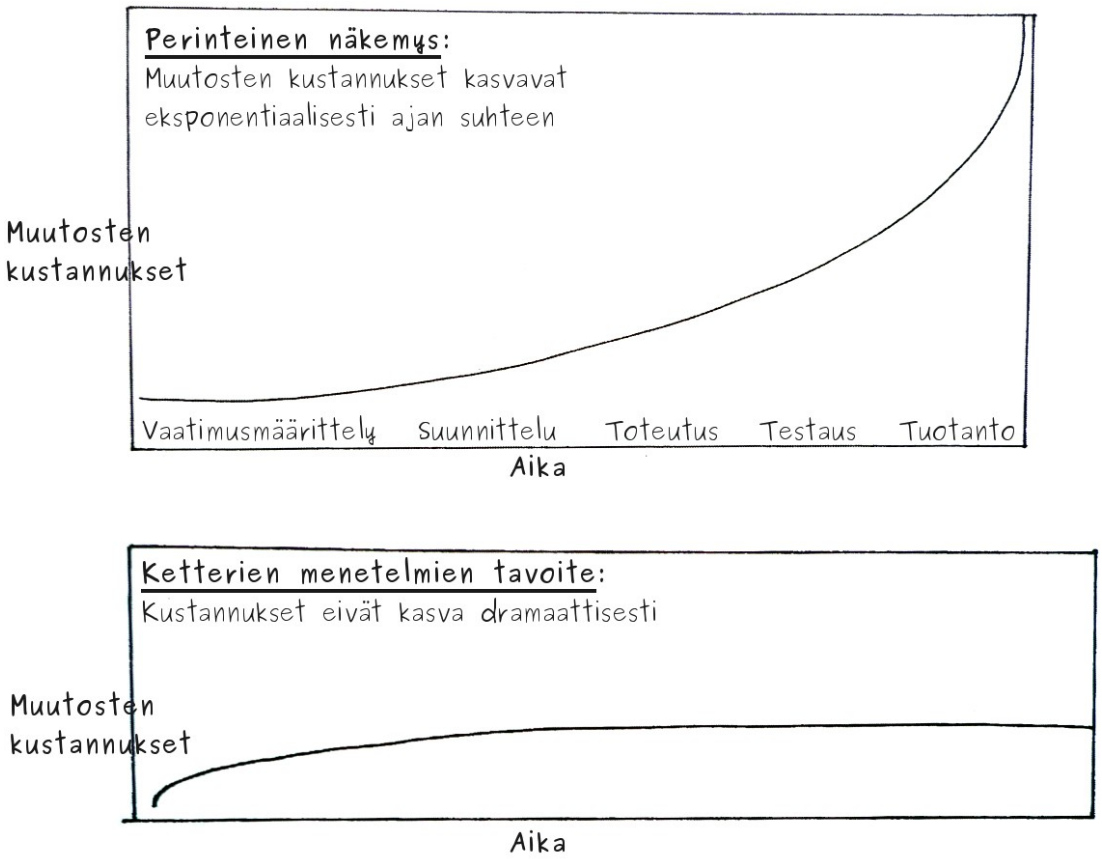
Vuosien mittaan evoluutiomallista on ollut käytössä useita eri versioita. Toiset menetelmät ovat sisältäneet vaatimusten määrittelyä ja dokumentointia etukäteen vesiputousmallin tapaan ja iteraatioiden kesto on saattanut olla useita kuukausia. Toiset ovat puolestaan painottaneet vähäistä etukäteen määrittelyä ja nopeita

muutaman viikon iteraatioita. Jälkimmäinen kuvaus sopii myös 1990-luvulla syntyneille ketterille menetelmille, joita tässä työssä esiteltävien Scrumin ja XP:n lisäksi ovat mm. Adaptive Software Development (ASD), Crystal, Dynamic Solutions Delivery Model (DSDM) ja Feature-Driven Development (FDD) [Abrahamsson02, luku 3]. Ketterät menetelmät ovat aina iteratiivisia ja inkrementaalisia, mutta iteratiiviset ja inkrementaaliset menetelmät eivät ole aina ketteriä.

Ketterät menetelmät pyrkivät ratkaisemaan seuraavat evoluutiomallin ongelmat: Kuten vesiputousmallin, myös evoluutiomallin suurin ongelma on muutosten teon vaikeus ja siitä aiheutuvat korkeat kustannukset. Muutosten teko koodiin on työlästä ja ohjelmiston rakenne korruptoituu, kun järjestelmän väliversioita muutetaan toistuvasti [Boehm88, Sommerville07, s. 69]. Usein evoluutiomallissa on myös ylläpidettäviä dokumentteja, jolloin muutosten teko aiheuttaa paljon lisätyötä, kun dokumentaatio päivitetään ajan tasalle [Dagnino02]. Turhia kustannuksia syntyy myös siitä, että pitkäkestoisilla iteraatioilla saadaan hitaammin palautetta tekeillä olevasta ohjelmistosta ja korjausliikkeet tehdään liian myöhään. Ketterien menetelmien keskeisin periaate on välttää muutosten korkeat kustannukset mahdollistamalla joustava muutosten teko projektin missä vaiheessa tahansa [Highsmith01, Larman07 s. 25].

Ketterät menetelmät olettavat, että vaatimuksia ei edes kannata selvittää kattavasti projektin alussa. Koska muutoksia tulee joka tapauksessa, on prosessit suunniteltu tukemaan muutosten tekoa. Kuvassa 2.3 ylhäällä on esitetty perinteinen käsitys muutosten teon kustannusten eksponentiaalisesta kasvamisesta projektin edetessä [Beck99b, luku 5]. Alapuolella puolella on käyrä, jota ketterät menetelmät tavoittelevat: kustannukset muutoksille eivät juuri kasva ajan mittaan. Ketterien menetelmien keinot muutosten hallintaan ovat korkealla tasolla seuraavat:

1. Muutosta tarvitsevien kohtien tunnistaminen mahdollisimman aikaisessa vaiheessa. Tämä tarkoittaa muutoksia sekä syntyvään ohjelmistoon että projektiryhmän työskentelytapoihin tai olosuhteisiin.



Kuva 2.3: Perinteinen näkemys kustannusten kasvamisesta ja ketterien menetelmien tavoite [Beck99b, luku 5].

2. Konkreettisten muutosten teon helpottaminen. Tämä tarkoittaa sekä ohjelmakoodia että dokumentaatiota.

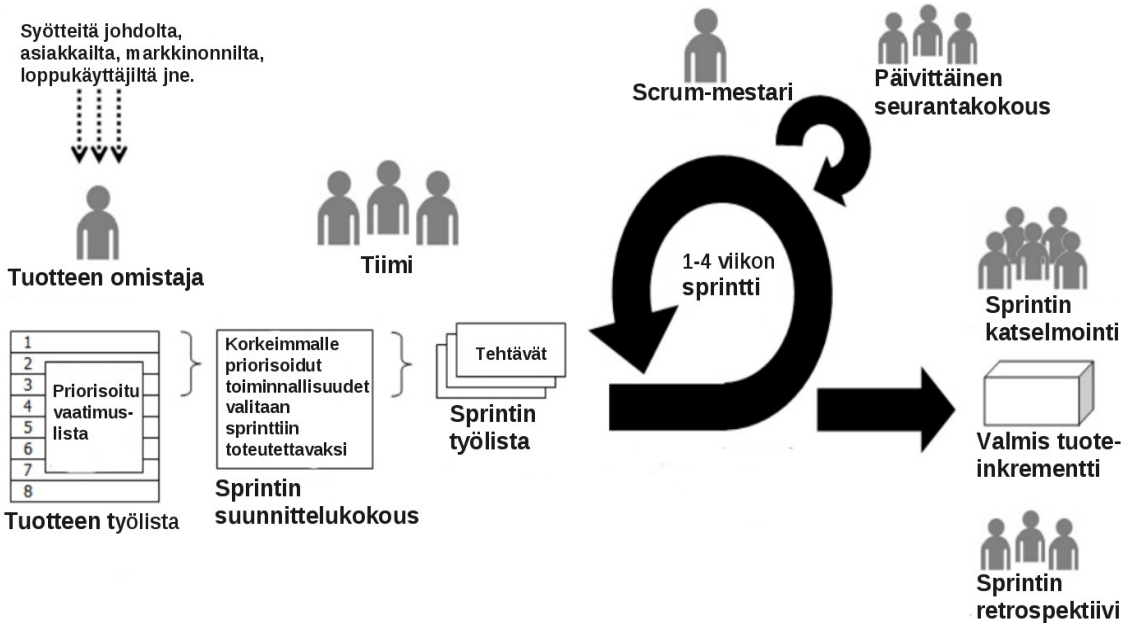
Muutosta tarvitsevat kohdat pyritään tunnistamaan iteraatioiden lyhyellä kestolla, joka on tyypillisesti yhdestä kuuteen viikkoa [Larman07, s. 267]. Näin palautetta syntyvästä ohjelmistosta saadaan nopeammin kuin kuukausia kestävissä iteraatioissa. Väärään suuntaan matkalla oleva projekti saadaan aikaisemmin takaisin raiteilleen ja turhalta työltä vältytään. Joissain prosessimalleissa asiakas tai loppukäyttäjä on antamassa palautetta myös iteraation sisällä. Tätä hyödyntävät mm. XP, Crystal ja DSDM [Abrahamsson02, luku 3]. Myös projektiryhmän työskentelystä pyritään saamaan tiheästi palautetta. Ketterät menetelmät korostavat ihmisten välistä vuorovaikutusta ja keskustelua, jonka avulla käsittelyä vaativat asiat tuodaan esille nopeasti ja niihin pyritään reagoimaan heti. Scrum-prosessimallin päivittäinen seurantakokous on käytännön esimerkki tästä periaatteesta (ks. luku 2.2).

Ohjelmakoodin muuttamista pyritään helpottamaan hyväksi havaituilla ohjelmointi-käytännöillä. Näitä ovat mm. testivetoinen ohjelmistokehitys ja automatisoidut testit. Dokumentaation määrä minimoidaan ja sen sijaan keskitytään tuottamaan toimivaa ja laadukasta ohjelmakoodia. Näihin menettelyihin tutustutaan tarkemmin luvussa 2.3, jossa esitellään Extreme Programming -prosessimalli ja sen hyödyntämät käytännöt konkreettisten muutosten teon kustannusten minimointiin.

2.2 Scrum

Scrum-prosessimalli [Schwaber01, Schwaber04] juontaa juurensa vuoteen 1986, jolloin Takeuchi ja Nonaka julkaisivat artikkelin menestyvien yritysten kuten Canonin ja Hondan uudesta iteratiivisesta tavasta tehdä onnistunutta tuotekehitystä [Takeuchi86]. Yhdysvalloissa 1990-luvun puolivälissä syntynyt ohjelmistotuotannon Scrum-menetelmä hyödyntää artikkelissa esiteltyjä käytäntöjä, kuten täysin itsenäisesti organisoituvia tiimejä. Ensimmäisen Scrum-projektin toteutti Jeff Sutherland yhdessä Jeff McKennan ja John Scumniotalesin kanssa Easel Corporation -ohjelmistoyrityksessä vuonna 1994 [Schwaber01]. Myöhemmin Sutherland ja Ken Schwaber laativat kuvauksen Scrum-prosessista ja se esiteltiin yleisölle OOPSLA-konferenssissa vuonna 1996. 2000-luvulla Scrumista on tullut yksi suosituimmista ketteristä menetelmistä.

Scrum-prosessimallin iteraation eli *sprintin (sprint)* keskeiset kohdat näytetään kuvassa 2.4. Sprintin syöte on *tuotteen työlista (product backlog)*, joka sisältää järjestelmän toiminnalliset ja ei-toiminnalliset vaatimukset, sekä kaiken työn, mitä järjestelmän toteutuksessa on vielä tehtävänä: toiminnot, korjaukset, parannukset ja muutokset [Schwaber01 s. 31-56, Schwaber04 s. s. 6-8]. *Sprintin suunnittelu-kokouksessa (sprint planning meeting)* tuotteen työlistalta valitaan seuraavassa iteraatiossa toteutettava toiminnallisuus, jonka pohjalta syntyy *sprintin työlista (sprint backlog)*. Valittu toiminnallisuus toteutetaan 1-4 viikkoa kestävän iteraation aikana. Iteraation jokaisena päivänä pidetään *päivittäinen seurantakokous (daily Scrum)*, jonka avulla tiimi pysyy ajan tasalla toistensa tekemisistä. Iteraation lopuksi toteutettu toiminnallisuus esitellään asiakkaalle *sprintin katselmoinnissa (sprint review)*. Seuraava iteraatio aloitetaan uudella sprintin suunnittelukokouksella.



Kuva 2.4: Scrum-prosessimalli [Sutherland07].

Tarkastellaan yksityiskohtaisemmin, mitä sprintti pitää sisällään: millaisia rooleja projektiryhmään kuuluu, mitä he tekevät ja mitä kunkin roolin vastuulla on. Scrum määrittelee kolme erilaista roolia: *tuotteen omistajan (product owner), Scrum-mestarin (Scrum master) ja tiimin.* Tuotteen omistaja edustaa sekä asiakasta että loppukäyttäjiä. Hän on vastuussa siitä, että projektin liiketoiminnallinen kannattavuus on mahdollisimman hyvä *(return on investment, ROI).* Hän pyrkii tähän priorisoimalla tuotteen työlistan siten, että eniten lisäarvoa tuottavat ominaisuudet toteutetaan ensimmäisenä.

Tuotteen työlistan sisältö voi tulla mistä vain, esimerkiksi loppukäyttäjiltä, markkinointiosastolta, johdolta, tuotteen omistajalta itseltään tai tiimiltä. Kukaan muu kuin tuotteen omistaja ei kuitenkaan saa priorisoida työlistaa. Tiimi auttaa tuotteen omistajaa listan priorisoinnissa arvioimalla työlistalla olevien ominaisuuksien toteutustyön määrää. Kuvassa 2.5 on esimerkki tuotteen työlistasta, jota käytettiin Scrum-prosessimallin projektinhallintaohjelmiston toteuttamisessa.

Tuotteen työlista				Sprintti ja jäljellä oleva työmäärä (henkilötyöpäivää)			
	Alkup.	Korjaus-	Korjattu	1	2	3	4
Vaatimus	arvio	kerroin	arvio	256	209	193	140
Projektin valinta tai uuden projektin luominen.	3	0,2	3,6	3,6	0	0	0
Työlista pohjaksi uudelle projektille	2	0,2	2,4	2,4	0	0	0
Luo tuotteen työlista muotoiluilla	3	0,2	3,6	3,6	0	0	0
Luo tuotteen työlista ilman muotoiluja	3	0,2	3,6	3,6	0	0	0
Näytä puunäkymä tuotteen työlistasta, julkaisuista ja sprinteistä	2	0,2	2,4	2,4	0	0	0
Sprintti 1 yhteensä	13	0,2	15,6	16	0	0	0
Luo uusi ikkuna tuotteen työlistan mallille	3	0,2	3,6	3,6	3,6	0	0
Luo uusi ikkuna sprintin työlistan mallille	2	0,2	2,4	2,4	2,4	0	0
Tuotteen työlistan Burndown-kaavio	5	0,2	6	6	6	0	0
Sprintin työlistan Burndown-kaavio	1	0,2	1,2	1,2	1,2	0	0
Näytä puunäkymä tuotteen työlistasta, julkaisuista ja sprinteistä	2	0,2	2,4	2,4	2,4	0	0
Näytä valitun sprintin tai julkaisun Burndown-kaavio	3	0,2	3,6	3,6	3,6	0	0
Sprintti 2 yhteensä	16	0,2	19	19	1,2	0	0
Arvojen ja summien automaattinen laskeminen	3	0,2	3,6	3,6	3,6	3,6	0
Kun työlistaa editoidaan erillisessä ikkunassa, päivitä pääikkunan graafia	2	0,2	2,4	2,4	2,4	2,4	0
Burndown-ikkunan piilotus ja automaattinen uudelleen näyttäminen	3	0,2	3,6	3,6	3,6	3,6	0

Kuva 2.5: Esimerkki tuotteen työlistasta [Schwaber04, s. 10].

Sprintin suunnittelukokouksessa tiimi arvioi, mitkä korkealle priorisoidut toiminnallisuudet se pystyy toteuttamaan seuraavan iteraation aikana. Nämä valitaan yhdessä tuotteen omistajan kanssa tuotteen työlistalta ja niiden toteuttaminen asetetaan sprintin tavoitteeksi. Tiimin vastuulla on valittujen toiminnallisuuksien toteuttaminen. Se organisoituu täysin itsenäisesti ja tekee työn niin kuin parhaaksi näkee.

Tiimi jakaa valitut ominaisuudet matalan tason työtehtäviin *(tasks)*, joille annetaan työmääräarviot tyypillisesti tunnin tarkkuudella. Tämän tuloksena syntyy sprintin työlista, josta nähdään esimerkki kuvassa 2.6. Projektiryhmä päivittää tehtävien jäljellä olevaa työmäärää listan sarakkeisiin, jotka kuvaavat sprintin päiviä. Listan työtehtäviä voidaan myös muokata kesken sprintin, jos tavoitteisiin pääseminen sitä vaatii. Tiimille on taattu työrauha: kukaan muu kuin tiimi ei saa koskea sprintin työlistaan, eikä tiimin tarvitse tehdä sprintin aikana mitään muuta kuin mihin se on suunnittelukokouksessa sitoutunut.

Sprintin työlista				Päivä ja jäljellä oleva työmäärä (tuntia)						
Tehtävä	Perustaja	Vastuuhenkilö	Status	1	2	3	4	5	6	7
Suunnittelupalaveri sprinttien 3-6 tavoitteista ja toiminnoista.	Danielle	Danielle / Sue	Valmis	20	0	0	0	0	0	0
Siirrä laskelmat pois Chrystal Reports -järjestelmästä	Jim	Allen	Ei aloitettu	8	8	8	8	8	8	8
Hae KEG-data		Tom	Valmis	12	0	0	0	0	0	0
Analysoi KEG-data – otsikko		Geaorge	Työn alla	24	24	24	24	12	10	10
Analysoi KEG-data – paketti		Tim	Valmis	12	12	12	12	12	4	4
Analysoi KEG-data – rasitteet		Josh	Työn alla							12
Analysoi KEG-data – kontakti		Danielle	Työn alla	24	24	24	24	12	10	8
Analysoi KEG-data – palvelut		Allen	Työn alla	24	24	24	24	24	12	10
Määritä ja luo tietokanta		Barry / Dave	Työn alla	80	80	80	80	80	80	60
Validoi KEG-tietokannan koko		Tim	Ei aloitettu							
Tarkista KEG-data levyllä G.		Dave	Työn alla	3	3	3	3	3	3	3
Arvojen ja summien automaattinen laskeminen		Sue	Ei aloitettu							

Kuva 2.6: Esimerkki sprintin työlistasta [Schwaber04, s.13].

Tiimin työntekoa synkronoi päivittäinen viidentoista minuutin mittainen seurantakokous. Kokouksessa jokainen tiimin jäsen vastaa lyhyesti seuraavaan kolmeen kysymykseen:

- Mitä olet tehnyt edellisen kokouksen jälkeen?
- Mitä teet seuraavaan kokoukseen mennessä?
- Hidastaako tai haittaako jokin työntekoasi?

Päivittäisen seurantakokouksen tarkoitus on pitää kaikki tiimin jäsenet ajan tasalla projektin senhetkisestä tilanteesta. Sen avulla asioihin voidaan reagoida heti ja mahdolliset työtä haittaavat tekijät korjataan nopeasti. Kokouksen järjestää Scrum-mestari. Scrum-mestarin vastuulla on järjestää tiimille parhaat mahdolliset työskentelyolosuhteet. Hän vastaa projektissa Scrumin käytäntöjen ja sääntöjen noudattamisesta. Hän auttaa sekä tiimiä että tuotteen omistajaa työskentelemään koko ajan mahdollisimman tuottavasti. Jos jokin estää tiimiä toimimasta maksimaalisella tehokkuudella, Scrum-mestarin tehtävä on poistaa tämä este. Tiimin jäsen saattaa esimerkiksi ilmoittaa johdon pyytävän häneltä jatkuvasti sprintin työlistan ulkopuolisia asioita. Tässä tilanteessa Scrum-mestari ohjeistaa häntä olemaan välittämättä pyynnöistä ja selittää johdolle, että tiimille on annettava työrauha.

Iteraation päätteeksi tiimi demonstroi toteutetun tuoteinkrementin asiakkaalle sprintin katselmoinnissa, jossa sekä tuoteinkrementtiä että projektin etenemistä

arvioidaan. Vain täysin tuotantoon valmis toiminnallisuus esitellään. Katselmoinnissa asiakkaan edustajat voivat esittää toivomiaan muutoksia tai uusia toiminnallisuuksia järjestelmään. Nämä lisätään tuotteen työlistaan tuotteen omistajalle priorisoitavaksi. Jos tiimiltä jäi jotain toiminnallisuutta toteuttamatta, ne siirretään seuraavaan sprinttiin.

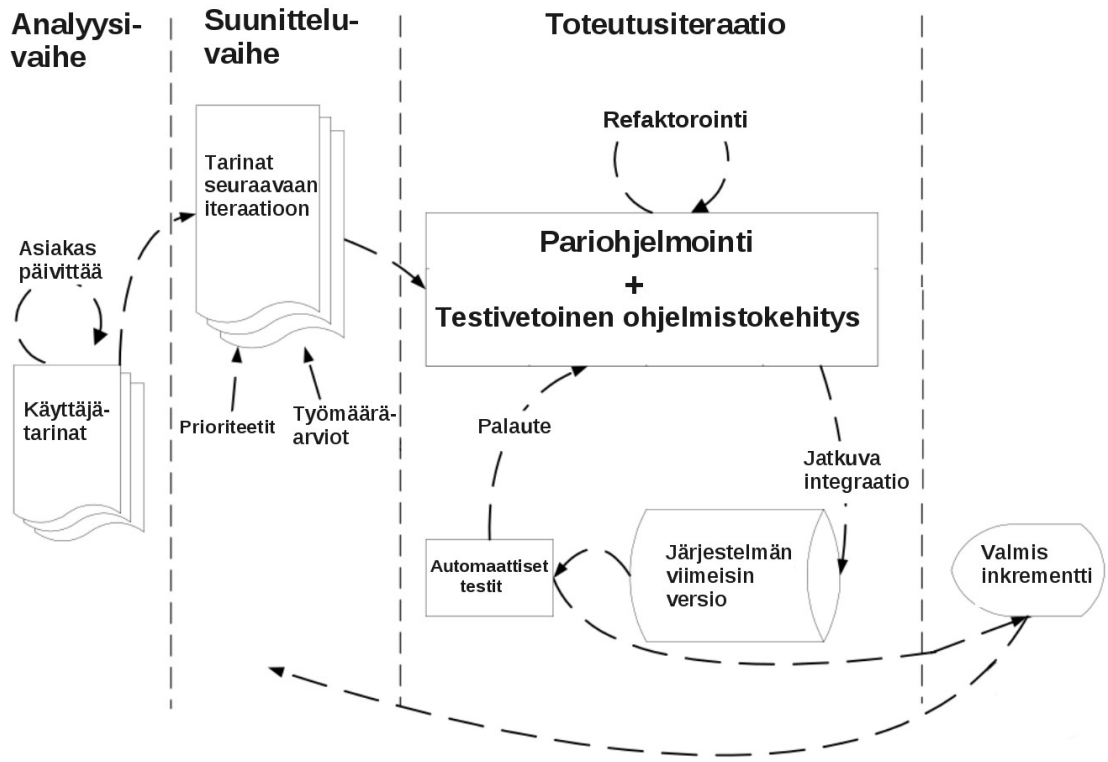
Iteraation lopuksi pidetään *sprintin retrospektiivi (sprint retrospective)*, jossa tiimi vastaa seuraaviin kysymyksiin: mitkä asiat menivät hyvin sprintin aikana ja mitä voitaisiin tehdä paremmin? Scrum-mestari auttaa tiimiä löytämään itse ratkaisut parannettaviin kohtiin. Näin tiimi pystyy parantamaan käytäntöjään iteraatio iteraatiolta. Sprintin katselmoinnin ja retrospektiivin jälkeen uusi iteraatio aloitetaan sprintin suunnittelukokouksella.

Scrum tarjoaa siis projektinhallinnallisen kehyksen, eikä se ota kantaa käytettyihin menetelmiin, kuten vaatimusten selvittämiseen tai ohjelmointikäytäntöihin. Scrum luottaa Barry Boehmin mainitsemaan tosiasiaan: ”Hyvät ammattitaitoiset ihmiset hyvällä arviointikyvyllä saavat projektit onnistumaan.” [Boehm91].

Scrum-prosessimallin käyttöönotosta on raportoitu hyviä kokemuksia. Suuressa yhdysvaltalaisessa 3M-yrityksessä Scrum paransi tiimien tuottavuutta huomattavasti [Moore07]. Nopeat iteraatiot ja päivittäiset seurantakokoukset minimoivat turhan työn määrää aiemmin käytettyyn vesiputousmalliin verrattuna. Joissain tapauksissa tiimit ovat pystyneet jopa neljä- viisinkertaiseen tuottavuuteen alan keskiarvoon nähden [Sutherland01]. Calgaryn yliopiston kaksi vuotta kestäneessä tapaus-tutkimuksessa Scrum vähensi toteuttajien ylitöitä oleellisesti ja lisäsi asiakkaiden tyytyväisyyttä [Mann05].

2.3 Extreme Programming

Extreme Programming (XP) syntyi Yhdysvalloissa 1990-luvun loppupuolella. XP on kokoelma Kent Beckin hyväksi havaitsemia ohjelmistotuotannon käytäntöjä [Beck99a]. XP:n iteraation keskeiset kohdat näytetään kuvassa 2.7. Iteraatioita



Kuva 2.7: Yleiskuva XP:n iteraatioiden kulusta (lähteen [Abrahamsson02] pohjalta editoitu).

edeltää lyhyt *analyysivaihe (exploration phase)*, jonka tuloksena syntyy asiakkaan kirjoittamia *käyttäjätarinoita (user stories).* Kuvassa 2.8 on esimerkkejä lento- ja hotellivarausjärjestelmän käyttäjätarinoista. Tarinat ovat siis samankaltaisia toimintokuvauksia kuin Scrum-menetelmän tuotteen työlistan toiminnot (kuva 2.5) ja asiakas määrittelee järjestelmän toiminnallisuuden niiden avulla. Tarinoiden täytyy olla sillä tasolla, että toteuttajat voivat antaa niistä työmääräarviot.

Suunnitteluvaiheessa (planning phase) asiakas valitsee toteutettavat käyttäjätarinat seuraavaan iteraatioon tarinoiden työmääräarvioiden ja prioriteettien perusteella. Ohjelmoijat jakavat tarinat matalan tason työtehtäviin *(tasks)*, jotka ovat vastaavia kuin Scrum-prosessimallin sprintin työlistassa (kuva 2.6). Iteraation aikana tehtävät toteutetaan pariohjelmoinnin avulla. Kun yhden käyttäjätarinan kaikki työtehtävät on toteutettu, verifioidaan näiden toimivuus ajamalla asiakkaan laatimat hyväksymis-testit. Iteraatio tuottaa valmiin inkrementin järjestelmästä ja seuraava iteraatio saa jälleen syötteekseen asiakkaan valitsemat käyttäjätarinat.

Näytä hotellit
Näytä paikan lähellä olevat hotellit.

Näytä hotellien saatavuus
Näytä hotellit, joissa on tilaa lentojen päivämäärillä.

Tarjoa edistyksellinen hotellihaku
Tarjoa asiakkaalle mahdollisuus hakea hotelleja myös muilla kriteereillä kuin päivämäärillä ja sijainnilla. Tämä sisältää varustelun, palvelun tason, kustannukset ja suositukset.

Hotellin varaus
Tee varaus. Veloita luottokorttia ja tarkista luottokortin kelpoisuus.

Kuva 2.8. Esimerkkejä lento- ja hotellivarausjärjestelmän käyttäjätarinoista [Beck00].

XP:n ohjelmointikäytännöt pyrkivät minimoimaan ohjelmakoodin ja dokumentaation muuttamisen aiheuttamia kustannuksia. Kuvan 2.7 toteutusiteraation menetelmistä keskeisimmät muutosten tekoa helpottavat ovat seuraavat [Beck99b]:

- ***Testivetoinen ohjelmistokehitys (test-driven development, TDD), jatkuva integraatio (continuous integration)* ja *automatisoidut testit.*** Yksikkötestit koodille kirjoitetaan ennen koodia ja tuotettu yksikkötestattu koodi integroidaan järjestelmään vähintään päivittäin. Kaikki järjestelmän testit ajetaan automaattisesti integroinnin yhteydessä. Kaikkien testien on mentävä aina läpi.
- **Koodin rakenne pidetään yksinkertaisena ja sitä *refaktoroidaan* jatkuvasti.** XP:ssä pyritään tekemään yksinkertaisin mahdollinen ratkaisu, jolla kaikki testitapaukset menevät läpi virheettä. Koodia refaktoroidaan jatkuvasti kehitystyötä tehdessä. Ohjelmoijat arvioivat aina ennen uuden toiminnallisuuden lisäämistä, voiko koodin rakennetta yksinkertaistaa. Jos voi, niin se tehdään heti.

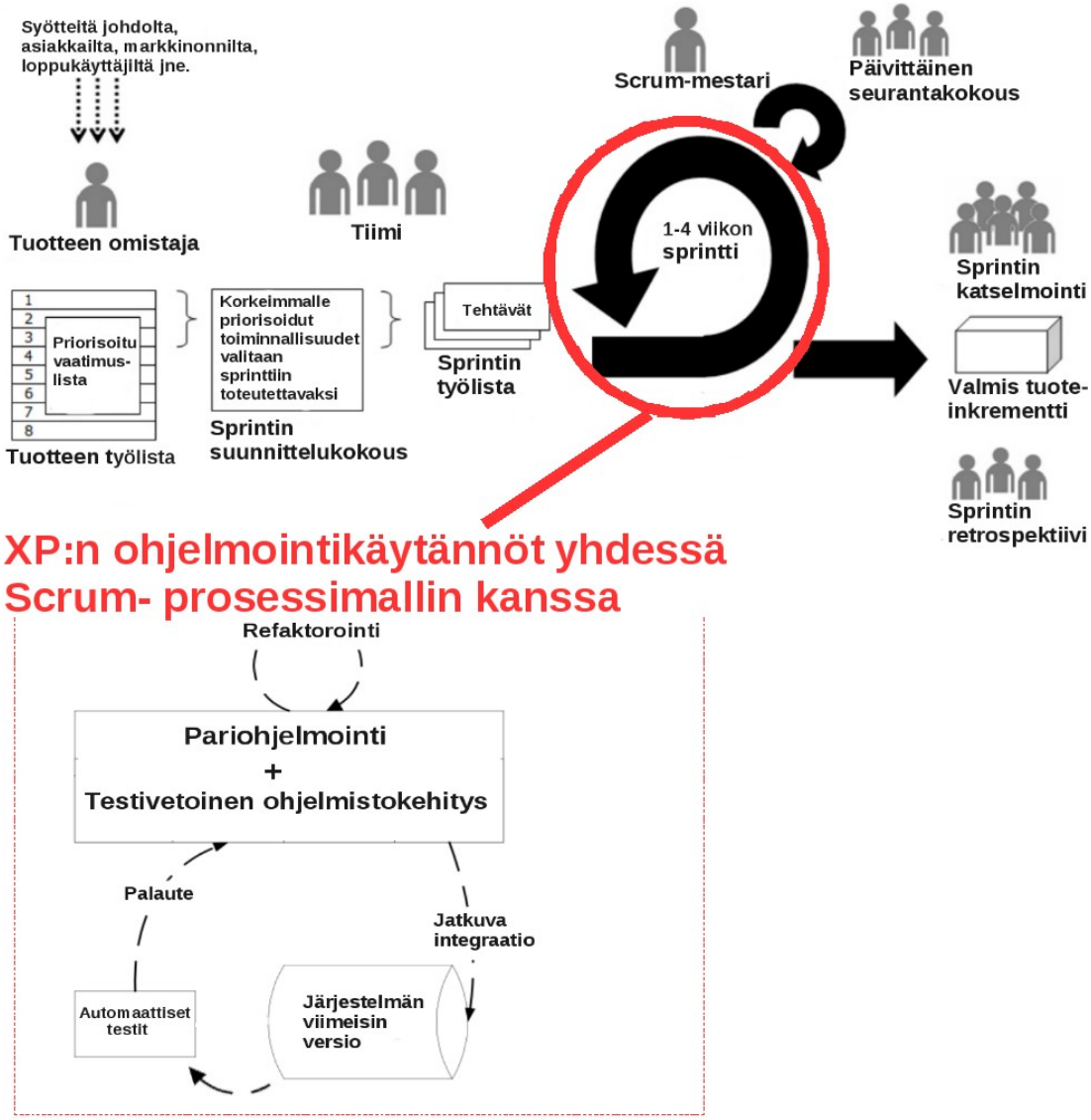
Yllä mainitut menettelyt tukevat muutosten tekoa. Kun koodin rakenne on yksinkertainen, sitä on helppo muuttaa. Muutoksia koodiin tehdään jatkuvasti refaktoroinnin yhteydessä, jolloin ohjelmoijat saavat rutiinia käytännön muutosten tekoon. Ohjelmoija voi tehdä muutoksia luottaen siihen, että automaattisesti ajetut testit saavat nopeasti kiinni mahdolliset muutoksen aiheuttamat ongelmat. Kirjoitetut

yksikkötestit sekä yksinkertainen ja luettava koodi toimivat teknisenä dokumentaationa, jolloin erillisen dokumentaation päivittämisestä ei tule lisätyötä. Menettelyt vastaavat osittain myös evoluutiomallin liian vähäiseksi jäävän testauksen ongelmaan [Dagnino02] ja siihen, että tuotettu prototyyppi ei ole teknisesti jatkokehitettävissä valmiiksi ohjelmistoksi [Boehm96].

Beckin esittämät menetelmät todetaan toimiviksi useissa tutkimuksissa. Capiluppi et al. seurasivat XP-projektia, jossa toiminnallisuutta lisättiin ja muutoksia tehtiin kahden ja puolen vuoden ajan [Capiluppi07]. Projektin tuottama koodi ei sisältänyt käytännössä yhtään monimutkaista osaa, vaikka perinteisesti on väitetty, että ohjelmiston monimutkaisuus kasvaa ja laatu heikkenee sitä mukaa kun uusia ominaisuuksia ja muutoksia lisätään [Parnas94]. XP:n käytännöt näyttävät lisäävän sekä ohjelmiston laatua että kehitysprosessin tuottavuutta [Maurer02, Moser08]. Pienemmässä kahden kuukauden XP-projektia seuranneessa tutkimuksessa muutosten teosta aiheutuneet kustannukset olivat vain 9,8 % kokonaiskustannuksista [Abrahamsson04].

2.4 Scrum ja Extreme Programming yhdistettynä prosessimallina

XP:n ja Scrum-prosessimallin iteraatioissa on paljon yhtäläisyyksiä. Extreme Programming tuottaa ohjelmoijien matalan tason työtehtävät asiakkaan priorisoimien käyttäjätarinoiden avulla. Scrum-menetelmässä vastaavat tehtävät laaditaan niinikään priorisoidun tuotteen työlistan perusteella, jonka sisältö on hyvin samankaltainen XP:n käyttäjätarinoiden kanssa. Scrum ei ota kantaa siihen, miten tehtävät toteutetaan. Extreme Programming puolestaan antaa tähän yksityiskohtaiset menetelmät. XP-prosessimallista puuttuu kuitenkin projektinhallinnallisia piirteitä, joita Scrumista löytyy. Näitä ovat esimerkiksi päivittäinen seurantakokous ja tiimiä auttava Scrum-mestari. Menetelmät täydentävät toisiaan ja niiden yhdistämisestä on saatu hyviä kokemuksia [Mar02]. Kuvassa 2.9 esitetään Extreme Programming -ohjelmointikäytännöt Scrum-prosessimalliin yhdistettynä.



Kuva 2.9: Extreme Programming -prosessimallin ohjelmointikäytännöt yhdessä Scrum-prosessimallin kanssa (lähteiden [Mar02] ja [Abrahamsson02] pohjalta editoitu) .

Kniberg ja Farhang saivat ratkaistua ruotsalaisen ohjelmistoyrityksen kriisiytyneen tilanteen Scrumin ja XP:n yhdistelmäprosessilla [Kniberg08]. Yrityksen olemassaolon kannalta kriittinen ohjelmisto oli kaatuilevaa monimutkaista spagettikoodia, joka saatiin vakaaksi ja yksinkertaisemmaksi XP:n ohjelmointi-menettelyillä. Useita ohjelmoijia loppuun polttanut työympäristö onnistuttiin muuttamaan normaaliksi. Lopulta projektin tuottavuus oli huomattavasti parempi kuin alkuperäisen, vaikka töitä teki vain puolet aiemmasta henkilöstömäärästä ja hekin noin puolet vähemmän työtunteja kuin aikaisemmin. Scrumin ja XP:n

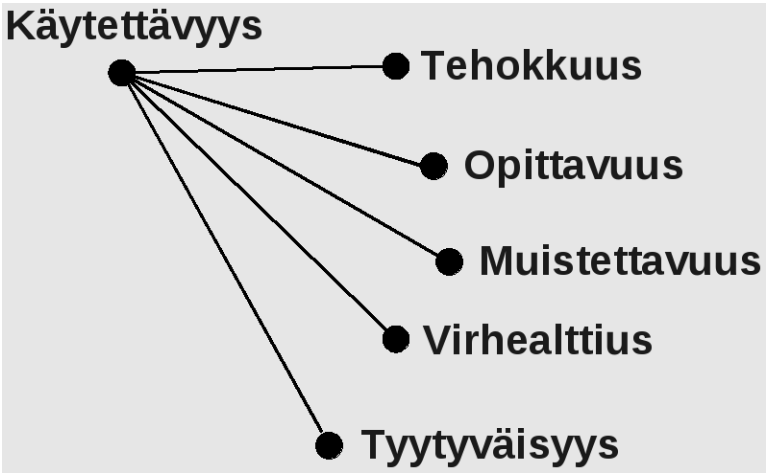
käytännöt auttoivat myös lentoyhtiö SAS:n ohjelmistotiimiä, joka paransi projektinhallintaansa ja ohjelmointikäytäntöjään niiden avulla [Jensen03]. Positiivisia kokemuksia raportoivat myös Salo ja Abrahamsson eurooppalaisille sulautettujen järjestelmien kehittäjille tehdyssä kyselytutkimuksessaan [Salo08].

3 GUIDe-prosessimalli ja simulointipohjainen GDD-käyttöliittymäsuunnittelu

Tässä luvussa esitellään GUIDe-prosessimalli ja simulointipohjainen GDD-käyttöliittymäsuunnittelumenetelmä. Luvussa 3.1 tutustutaan hyvän käyttöliittymän osa-alueisiin, hyödyllisyyteen ja käytettävyyteen. Yleiskuva GUIDe-prosessimallista esitellään luvussa 3.2. GUIDen ja GDD:n käytännöt esitellään yksityiskohtaisesti luvussa 3.3.

3.1 Hyödyllisyys ja käytettävyys

Hyödyllisyys (utility) ja *käytettävyys (usability)* määrittelevät käyttöliittymän soveltuvuuden käyttötarkoitukseensa [Nielsen93 s. 25, Laakso06a s. 6-11]. Tässä työssä hyödyllisyys tarkoittaa sitä, että järjestelmän toiminnallisuus ja tietosisältö mahdollistavat loppukäyttäjien käyttötilanteiden suorittamisen alusta loppuun asti. Kun käyttötilanteet pystyy suorittamaan käyttöliittymällä jotenkin, voidaan arvioida käytettävyyttä eli tekemisen sujuvuutta [Laakso06a, s. 10-11]. Jakob Nielsenin esittämät käytettävyyden osa-alueet ovat tehokkuus, opittavuus, muistettavuus, virhealttius ja tyytyväisyys (kuva 3.1) [Nielsen93, s. 25].



Kuva 3.1: Käytettävyyden osa-alueet [Nielsen93, s. 25].

Tehokkuudella tarkoitetaan tekemisen suoraviivaisuutta. Se voidaan jakaa *mekaaniseen tehokkuuteen* ja *mentaaliseen tehokkuuteen* [Laakso06c, s. 8-11]. Mekaanista tehokkuutta ovat esimerkiksi klikkaukset ja ikkunasiirtymät. Jos käyttäjä joutuu tekemään paljon turhia toimenpiteitä saadakseen työtehtävän hoidettua, aiheutuu turhasta navigoinnista mekaaninen tehokkuusongelma. Turhat toimenpiteet voivat olla esimerkiksi edestakaisin navigointia eri näkymien välillä. Jos käyttäjä joutuu pitämään mielessään eri näkymien tietoja, aiheutuu käyttäjän rajallisen *työmuistin* (*short-term memory* [Norman88, s. 66-67]) kuormittumisesta mentaalinen tehokkuusongelma. Mentaaliseen tehokkuuteen laskettavaa turhaa ajatustyötä on tietojen mielessä pitämisen lisäksi esimerkiksi sellaisten lukujen päässä laskeminen, jotka järjestelmä voisi laskea käyttäjälle valmiiksi.

Tehokkuudeltaan hyvä järjestelmä näyttää tarvittavat tiedot ja toiminnot käyttäjälle yhdellä kertaa, jolloin turhalta navigoinnilta ja tietojen mielessä pitämiseltä voidaan välttyä. Käyttöliittymän huono tehokkuus aiheuttaa usein isoja ongelmia loppu-käyttäjien työnteolle: Gulliksen ja Sandblad kohtasivat tutkimuksessaan terveyden-huollon järjestelmiä, joissa työtehtäviä suorittavien käyttäjien ajasta 80 % kului turhaan navigointiin käyttöliittymässä [Gulliksen95, s. 128]. Borälv et al. analysoivat terveydenhuollon työtilanteita ja havaitsivat tietokonejärjestelmästä aiheutuvan turhan työmuistin kuormittumisen vievän käyttäjän kognitiivista kapasiteettia pois työtehtävässä tarvittavasta ongelmanratkaisusta [Borälv94, luku 2].

Opittavuudella tarkoitetaan sitä, että käyttäjä keksii itse, mistä hän saa tehtyä haluamansa toimenpiteen ja mitä käyttöliittymän tiedot tarkoittavat. Käyttöliittymän *muistettavuus* on hyvä, kun käyttäjä kerran opittuaan osaa käyttää ohjelmistoa. Jos järjestelmässä on hyvä opittavuus, ei muistettavuudesta tarvitse huolehtia: jos käyttäjä ei muista miten ohjelmaa käytetään, on se joka tapauksessa helppo oppia uudestaan. *Virhealttius* määrittelee sen, kuinka usein käyttäjät tekevät virheitä ja kuinka he selviytyvät niistä. Hyvä opittavuus vähentää käyttäjien mahdollisia virheaskelia. Myös hyvä tehokkuus karsii virheitä, sillä käyttäjän suoraviivainen tukeminen ilman turhia toimenpiteitä vähentää potentiaalisia kohtia virheaskelille. Työtehtävän suorittamiseen 24 toimenpideaskelta vaativassa järjestelmässä on kolme kertaa niin paljon mahdollisia kohtia tehdä virhe kuin 8 askelta vaativassa

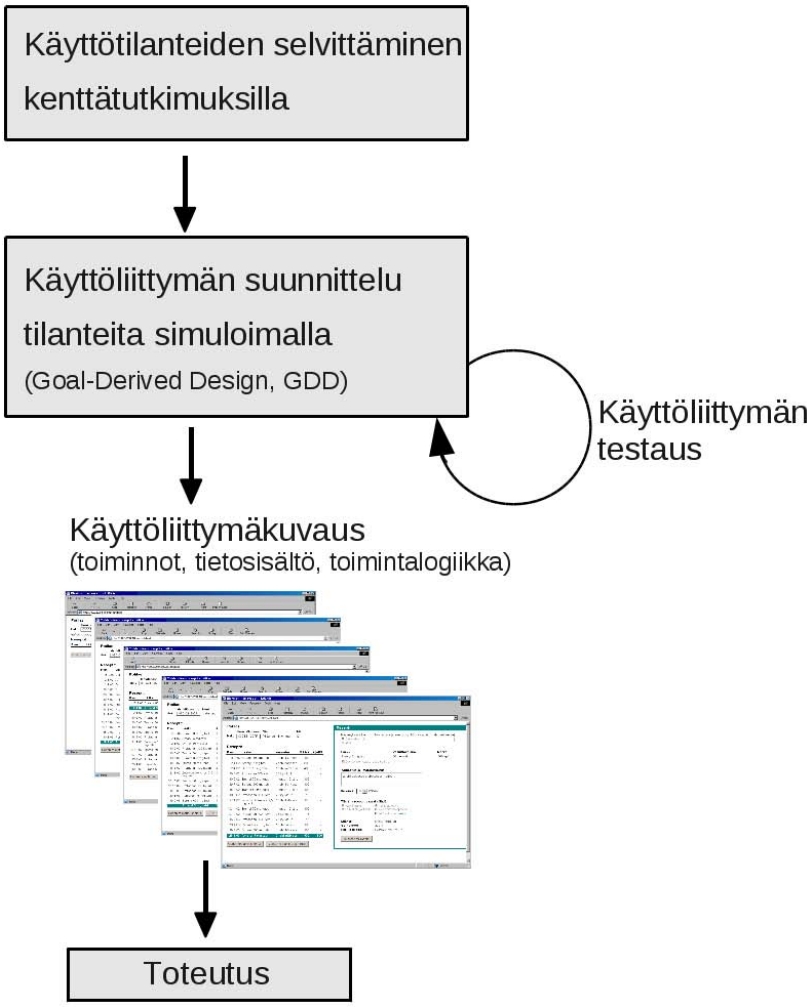
järjestelmässä. Käytettävyyden osa-alueiden viimeinen kohta eli *tyytyväisyys* tarkoittaa järjestelmän käytön subjektiivista miellyttävyyttä.

3.2 GUIDe-prosessimalli

Ohjelmiston käyttöliittymä on yksi keskeisimpiä ohjelmistotuotannon riskikohtia. Yleisiä toteutuvia riskejä ovat sekä ohjelmiston väärä toiminnallisuus ja tietosisältö että loppukäyttäjän työtehtävien kannalta epäoptimaalinen käyttöliittymä [Boehm91]. Tehokas menettely näiden riskien minimointiin on käyttöliittymäprototyyppien laatiminen [Boehm91, Bäumer96, Gomaa81]. Prototyypin avulla järjestelmän soveltuvuutta käyttötarkoitukseensa voidaan testata jo projektin alkuvaiheessa [Bäumer96].

GUIDe-prosessimallissa *(Goals – User Interface Design – Implementation)* projekti aloitetaan kenttätutkimuksilla, joilla selvitetään loppukäyttäjien konkreettiset käyttötilanteet käyttöliittymäsuunnittelun syötteeksi (kuva 3.2) [Laakso04a]. Käyttöliittymä suunnitellaan simuloimalla selvitettyjä käyttötilanteita *Goal-Derived Design* -menetelmällä (*GDD*), joka on GUIDen ydin. Suunnittelun tuloksena syntyy näyttökuvista koostettu prototyyppi, josta järjestelmän toiminnot, tietosisältö ja interaktiotavat selviävät. Syntynyttä prototyyppiä voidaan testata useilla eri menetelmillä sekä loppukäyttäjien kanssa että ilman (ks. luku 3.3.4). Testauksessa ilmenneet ongelmat korjataan ennen toteutusvaiheeseen siirtymistä.

GUIDen lisäksi kirjallisuudesta löytyy muutamia samankaltaisia käytäntöjä hyödyntäviä käyttöliittymälähtöisiä vaatimusmäärittelymenetelmiä. Näitä ovat esimerkiksi Lauesenin ja Harningin *Virtual Windows* [Lauesen01], Beyerin ja Holtzblattin *Contextual Design* [Beyer99], sekä Cooperin *Goal-Directed Design* [Cooper03]. Kaikille mainituille menetelmille on yhteistä loppukäyttäjien työn-kulkujen selvittäminen haastatteluilla ja käyttäjän työn tarkkailemisella sekä käyttäjäselvitysten tulosten hyödyntäminen käyttöliittymäsuunnittelun perustana.



Kuva 3.2: GUIDe-prosessimalli (lähteen [Laakso04a] pohjalta editoitu).

Isoja eroja on siinä, millä tavalla haastatteluista ja tarkkailuista kerätty materiaali tulkitaan ja jäsennetään käyttöliittymäsuunnittelun syötteeksi ja miten käyttöliittymä suunnitellaan käyttäjäselvitysten tulosten perusteella. Esimerkiksi Contextual Design -menetelmässä käyttäjien työtehtäviä mallinnetaan usealla erilaisella tavalla [Beyer98], mutta menettelyt hyvän käyttöliittymän suunnitteluun jätetään auki [Beyer98, s. 379]. GUIDessa sen sijaan pyritään selvittämään konkreettisia esimerkkejä loppukäyttäjille eteen tulevista tilanteista ja käyttöliittymä suunnitellaan GDD-menetelmällä simuloimalla yhtä tilannetta kerrallaan toimenpide toimenpiteeltä. Suunnittelija piirtää tietoja ja toimintoja näkyville sitä mukaa kuin niitä simuloinnin edetessä tarvitaan, tavoitellen koko suunnittelun ajan loppu-käyttäjän kannalta parasta mahdollista ratkaisua.

GDD tuottaa sellaisen paperiprototyypin, jolla suunnittelussa syötteenä olleiden käyttötilanteiden suorittaminen järjestelmällä voidaan simuloida askel askeleelta alusta loppuun asti. Suunnitteluvaiheen simuloinnin ja prototyypin testaamisen avulla voidaan välttää vastaavat tilanteet kuin luvussa 3.1 mainittu tapaus, jossa valmiilla järjestelmällä työtä tekevät käyttäjät kuluttivat jopa 80 % ajastaan turhaan navigointiin käyttöliittymässä. Kyseisessä tapauksessa ongelmat olivat tulleet esille vasta käyttöönoton jälkeen, GUIDen ja GDD:n avulla riskit ongelmallisen järjestelmän päätymisestä edes toteutusvaiheeseen voidaan minimoida.

3.3 GUIDen ja GDD:n vaiheet

Tässä luvussa esitellään GUIDen ja GDD:n vaiheet, joiden havainnollistamiseksi katsotaan esimerkkejä elektronisen reseptin kehitysprojektista, jossa Interacta Design Oy suoritti kenttätutkimuksia ja teki käyttöliittymäsuunnittelua. Luvussa 3.3.1 tarkastellaan GUIDen käyttäjäselvitysmenetelmiä. Luvussa 3.3.2 käsitellään käyttöliittymän suunnittelua GDD-menetelmällä. Suunnittelussa syntyneen käyttöliittymäratkaisun dokumentointi näyttökuvien avulla esitellään luvussa 3.3.3 ja näyttökuvista koostuvan prototyypin testausmenetelmät luvussa 3.3.4.

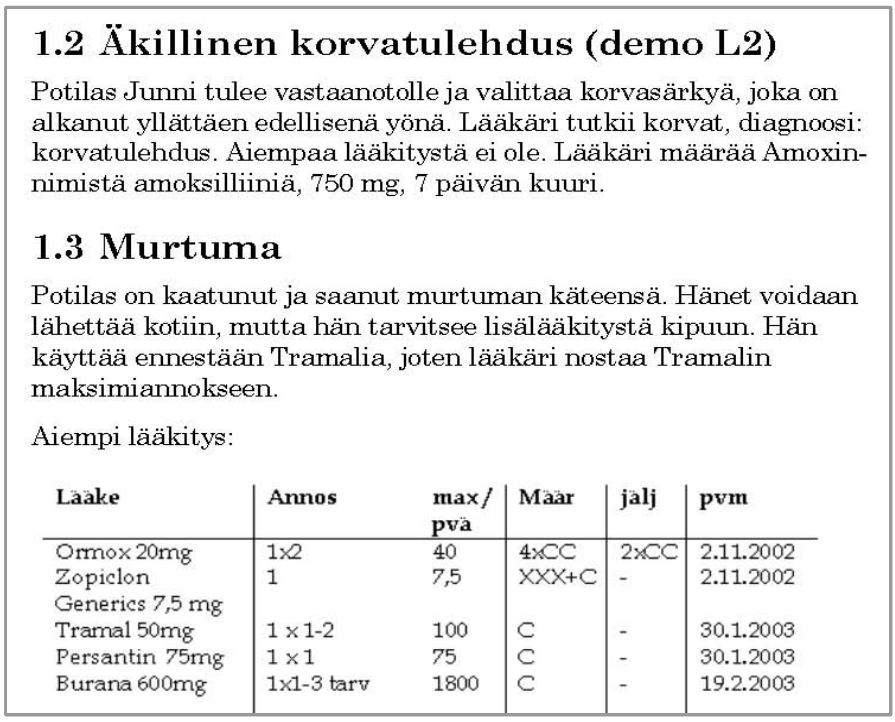
3.3.1 Konkreettisten käyttötilanteiden selvittäminen kenttätutkimuksilla

Jotta simulointipohjainen suunnittelu on mahdollista, täytyy loppukäyttäjille eteen tulevat konkreettiset *käyttötilanteet* selvittää simuloinnin syötteeksi [Laakso06a, s. 13-16, s. 45]. Tämä tehdään *kenttätutkimuksilla (field studies, site visits)* [Hackos98]. Kaksi keskeisintä GUIDe-prosessimallissa käytettävää kenttätutkimusmenetelmää ovat *käyttäjätarkkailut (user observations)* ja *kontekstuaaliset haastattelut (contextual interviews)* [Laakso04a]. Molempien menetelmien tavoite on sama: selvittää, millaisia tilanteita loppukäyttäjille tulee eteen ja miten he hoitavat ne. Millaisia olemassa olevia työnkulkuja käyttäjillä on? Mitä järjestelmiä, lomakkeita, muistiinpanoja tai muita tietoja he tarvitsevat ja miten he näitä käyttävät? Ilman loppukäyttäjien työn yksityiskohtaista tuntemista on mahdotonta suunnitella järjestelmää, joka tukisi heidän työtään optimaalisesti [Hackos98, luku 1].

Sekä tarkkailuissa että kontekstuaalisissa haastatteluissa käyttäjiä mennään tapaamaan heidän toimintaympäristöönsä. Käyttäjätarkkailuissa loppukäyttäjän toimintaa tarkkaillaan hänen suorittaessaan työhön tai vapaa-aikaan liittyviä tehtäviään [Diaper89]. Tarkkailija ei häiritse käyttäjän työskentelyä vaan keskittyy tekemään muistiinpanoja. Tarvittavat tarkentavat kysymykset kysytään vasta työtehtävien suorittamisen jälkeen.

Jos käyttäjän työtä ei ole mahdollista tarkkailla, yritetään käyttäjälle eteen tuleviin tilanteisiin päästä kiinni kontekstuaalisten haastattelujen avulla. Esimerkiksi elektronisen reseptin järjestelmän käyttäjäselvityksissä voi olla vaikea päästä lääkärin vastaanotolle tarkkailemaan potilaan diagnosointia ja reseptien määräämistä. Haastatteluja kannattaa käyttää myös silloin, kun suunniteltavan järjestelmän käyttöön osuvia tilanteita tulee eteen harvoin tai koko työnkulun näkeminen kestäisi pitkän aikaa, esimerkiksi kuukausia. Kuten tarkkailujen, myös kontekstuaalisten haastattelujen tarkoituksena on nähdä kuinka käyttäjä tekee työtään [Beyer98, s. 64-66]. Haastattelija pyytää haastateltavaa näyttämään, miten hän hoitaa eteen tulevia tilanteitaan. Toisin kuin tarkkailuissa, nyt konkreettisen työn näkeminen voi olla haastavaa. Käyttäjät puhuvat asioista mielellään abstraktilla tasolla, joka ei ole hyödyllistä käyttöliittymäsuunnittelulle [Beyer98, s. 47-51]. Tällöin on haastattelijan vastuulla, että yleisellä tasolla keskustelun sijaan käyttäjä näyttää askel askeleelta, kuinka hän tekee työtään nykyisillä menettelyillä.

Jotta konkreettisten käyttötilanteiden laatiminen olisi mahdollista, pyritään tarkkailuissa ja haastatteluissa saamaan talteen kaikki käyttäjän työtehtäviin liittyvä data [Hackos98, s. 262-263]. Yleistyksien ja yhteenvetojen sijaan on tärkeää, että esille tulevat tilanteet ja niihin liittyvät asiat saadaan ylös konkreettisesti [Beyer98, s. 47-51]. Esimerkkejä lääkärin kontekstuaalisessa haastattelussa selvitetyistä käyttötilanteista näytetään kuvassa 3.3. Esimerkiksi jos tilanteen *1.3 Murtuma* sijaan haastattelun tuloksena olisi ylimalkainen tilanne ”Lääkäri määrää potilaalle Tramalia kipuun.”, puuttuisi tilanteesta käyttöliittymäsuunnitteluvaiheen simuloinnissa tarvittavaa konkretiaa. Kuvan 3.3 tilanteessa *1.3 Murtuma* esitetään potilaan aiempi lääkitys sekä tieto siitä, että aiempi lääkitys vaikuttaa lääkärin päätökseen määrättävästä lääkityksestä. Nyt käyttöliittymäsuunnittelija pystyy ottamaan



1.2 Äkillinen korvatulehdus (demo L2)

Potilas Junni tulee vastaanotolle ja valittaa korvasärkyä, joka on alkanut yllättäen edellisenä yönä. Lääkäri tutkii korvat, diagnoosi: korvatulehdus. Aiempaa lääkitystä ei ole. Lääkäri määrää Amoxin-nimistä amoksilliiniä, 750 mg, 7 päivän kuuri.

1.3 Murtuma

Potilas on kaatunut ja saanut murtuman käteensä. Hänet voidaan lähettää kotiin, mutta hän tarvitsee lisälääkitystä kipuun. Hän käyttää ennestään Tramalia, joten lääkäri nostaa Tramalin maksimiannokseen.

Aiempi lääkitys:

Lääke	Annos	max/ pvä	Määr	jälj	pvm
Ormox 20mg	1x2	40	4xCC	2xCC	2.11.2002
Zopiclon Generics 7,5 mg	1	7,5	XXX+C	-	2.11.2002
Tramal 50mg	1 x 1-2	100	C	-	30.1.2003
Persantin 75mg	1 x 1	75	C	-	30.1.2003
Burana 600mg	1x1-3 tarv	1800	C	-	19.2.2003

Kuva 3.3: Lääkärin kontekstuaalisella haastattelulla selvitettyjä käyttötilanteita [Interacta03].

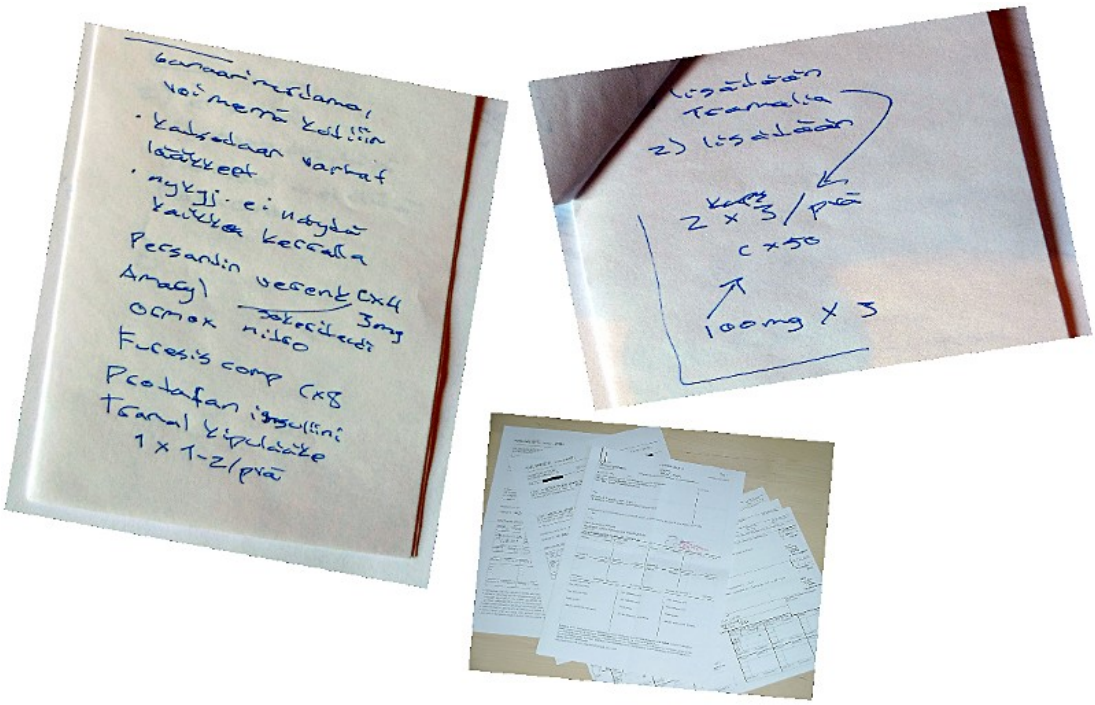
simuloinnissa huomioon, että lääkäri tarvitsee potilaan aiemman lääkityksen suoraviivaisesti näkyville käyttöliittymään. Ylimalkaisella tilanteella suunnitellessa on riski, että potilaan aiempi lääkitys päätyy pidemmän polun päähän järjestelmässä. Tästä seuraisi lääkärille turhaa navigointia käyttöliittymässä eli tehokkuusongelma.

Tyypillisesti riittävä menettely tarvittavan konkretian saavuttamiseen on kirjoittaa muistiinpanot ylös joko kynällä paperille tai kannettavalle tietokoneelle [Diaper89, s. 221-222], sekä ottaa digikuvia nykyisestä työnkulusta [Laakso06b]. Lisäksi kaikista työtehtäviin liittyvistä materiaaleista, kuten muistilapuista ja lomakkeista otetaan kopiot [Hackos98, s. 138-140]. Kuvassa 3.4 näytetään digikuvia käyttäjätarkkailusta apteekissa, jossa ollaan seuraamassa reseptien käsittelyä. Digikuvat helpottavat työnkulun ja tilanteen palauttamista mieleen tarkkailun jälkeen.

Kuvassa 3.5 on lääkärin kontekstuaalisesta haastattelusta kerättyä materiaalia. Haastattelija on pyytänyt lääkäriä käymään läpi aiemmin päivällä kohtaamiaan potilaita ja näyttämään, millä perusteella ja miten hän kirjoitti heidän reseptinsä. Tuloksena tästä on kuvan mukaisia yksityiskohtaisia muistiinpanoja. Lisäksi

Kuva 3.4: Digikuvia reseptien käsittelystä apteekissa [Interacta03].

haastattelija on ottanut kopioita lääkärin kirjoittamista resepteistä. Kopiot oikeista resepteistä antavat realistista esimerkkidataa käyttöliittymäsuunnitteluun ja auttavat ymmärtämään lääkärin työtä: mitkä reseptilomakkeen kohdat lääkäri täyttää ja mitä hän kirjoittaa niihin [Hackos98, s. 139].



Kuva 3.5: Lääkärin kontekstuaalisesta haastattelusta kerättyä materiaalia: haastattelumuistiinpanoja ja kopioita kirjoitetuista resepteistä [Interacta03].

Käyttäjän luona vierailun jälkeen nykyiset työnkulut on siis tallennettu muistiinpanoilla, digikuvilla ja materiaalikopioilla. Kuvan 3.3 kaltaisten käyttötilanteiden laatimista varten nykyisestä työnkulusta irrotetaan käyttäjän tavoite: mitä hän yrittää tehdä ja minkä takia [Hackos98, s. 50-60]? Käyttötilanteiden

perusidea on sama kuin Hackosin ja Redishin *lyhyen skenaarion (brief scenario)*: ”Lyhyt skenaario kuvailee tilanteen, jonka käyttäjän on pystyttävä hoitamaan järjestelmän avulla. Se ei kerro mitään siitä, miten käyttäjä hoitaa tilanteen nykyisellä järjestelmällä tai miten tilanne tullaan hoitamaan uudella järjestelmällä. Tilanne pysyy aina samana, vaikka sen ratkaisutapaa muutettaisiin useilla eri tavoilla.” [Hackos98. s. 324].

Esimerkiksi kuvan 3.3 tilanteessa *1.3 Murtuma* sekä potilaan murtuma ja aiempi lääkitys että lääkärin diagnoosi ja päätös lääkityksestä pysyvät samoina, vaikka tilanteen ratkaisutapaa muutetaan. Nykymenettelyssä lääkäri kirjoittaa Tramal-reseptin paperille ja lisäksi tiedot lääkityksestä työpaikkansa järjestelmään. Potilas vie paperireseptin apteekkiin, jossa virkailija kopioi sen tiedot manuaalisesti apteekin järjestelmiin mm. Kela-laskutuksia varten. Menettelyssä on paljon päällekkäistä työtä. Saman tilanteen voisi hoitaa myös siten, että lääkäri kirjoittaa reseptin kaikille osapuolille yhteiseen elektronisen reseptin järjestelmään. Potilas saa lääkkeet apteekista henkilötiedoillaan ja tiedot ovat valmiina järjestelmässä myös muita osapuolia, kuten Kelaa varten. Päällekkäistä työtä ei tehdä. Käyttötilanne pysyy samana, vaikka kaksi esiteltyä ratkaisutapaa ovat täysin erilaiset.

Käyttötilanteista voidaan edelleen analysoida *tavoitepohjaisia käyttötapauksia*, jotka virittävät ratkaistavan ongelman käyttöliittymäsuunnittelijoille jäsennellymmin ja yksityiskohtaisemmin kuin käyttötilanteet. Tavoitepohjaiset käyttötapaukset erittelevät käyttötapauksen nykytilan ja tavoiteltavan lopputuloksen, joiden välillä vallitseva ristiriita pitäisi ratkaista suunnitteilla olevan järjestelmän avulla [Laakso05, s. 42-51]. Vaikka pelkillä käyttötilanteilla ei saada edellä mainittuja etuja, voi simulointipohjaista suunnittelua tehdä niiden perusteella ja ne riittävät tämän työn kannalta GDD-menetelmän ymmärtämiseen. Siten tavoitepohjaisia käyttötapauksia ei käsitellä tässä työssä yksityiskohtaisesti.

3.3.2 Käyttöliittymäratkaisun laatiminen simuloimalla

Käyttöliittymäratkaisu laaditaan kenttätutkimuksissa selvitettyjä käyttötilanteita simuloimalla [Laakso04a]. Kantavana ajatuksena on jo suunnitteluvaiheessa simuloida järjestelmän todellista käyttöä, jota vasten millä tahansa menetelmällä suunnitellun järjestelmän käyttökelpoisuutta viime kädessä arvioidaan [Laakso06a, s. 43-44]. Selvitetyt käyttötilanteet priorisoidaan suunnittelua varten siten, että tyypilliset usein toistuvat tilanteet saavan korkeamman prioriteetin kuin epätyypillisemmät tilanteet.

GDD-suunnittelumenetelmän vaiheet näytetään kuvassa 3.6. Aluksi suunnittelija ottaa korkeimmalle priorisoidun käyttötilanteen ja piirtää tarvittavat tiedot ja toiminnot näkyviin simuloimalla käyttäjän toimintaa askel askeleelta (kuvassa 3.6 käyttötilanteen 1 vaihe 1.1). Tilanteelle pyritään aina kiinnittämään tehokkuuden kannalta optimaalisin ratkaisu [Laakso05, s. 39-41]. Tämän takia kaikki tiedot ja toiminnot piirretään yhteen suureen ikkunaan, jotta käyttäjä voi hoitaa tehtävänsä ilman turhia askelia ja tietojen mielessä pitämistä.

Toimenpide a. Piirrä yksi käyttötilanne kerrallaan dataa ja toimintoja näkyviin 'yhteen suureen ikkunaan'.

Käyttötilanne 1
1.1 Piirrä kt 1 käyttöliittymäratkaisuksi simuloimalla käyttösekvenssiä vaihe vaiheelta.
1.2 Simuloi ilmeisimmät variaatiot.

Käyttötilanne 2
2.1 Editoi kt 2 edelliseen käyttöliittymään mukaan simuloimalla sitä vaihe vaiheelta.
2.2 Simuloi ilmeisimmät variaatiot.
2.3 Testaa käyttöliittymää simuloimalla edellinen käyttötilanne 1.

Käyttötilanne 3
3.1 Editoi kt 3 mukaan simuloimalla.
3.2 Simuloi ilmeisimmät variaatiot.
3.3 Testaa käyttöliittymää simuloimalla edelliset käyttötilanteet 1 ja 2.

Käyttötilanne 4
...

Toimenpide b. Testaa käyttötilanneketjut (vain mielekkäät tositilanteet).
Simuloi esim. ketju 2, 1, 2, 4, 2, 2, 4.
Simuloi esim. ketju 1, 1, 3.

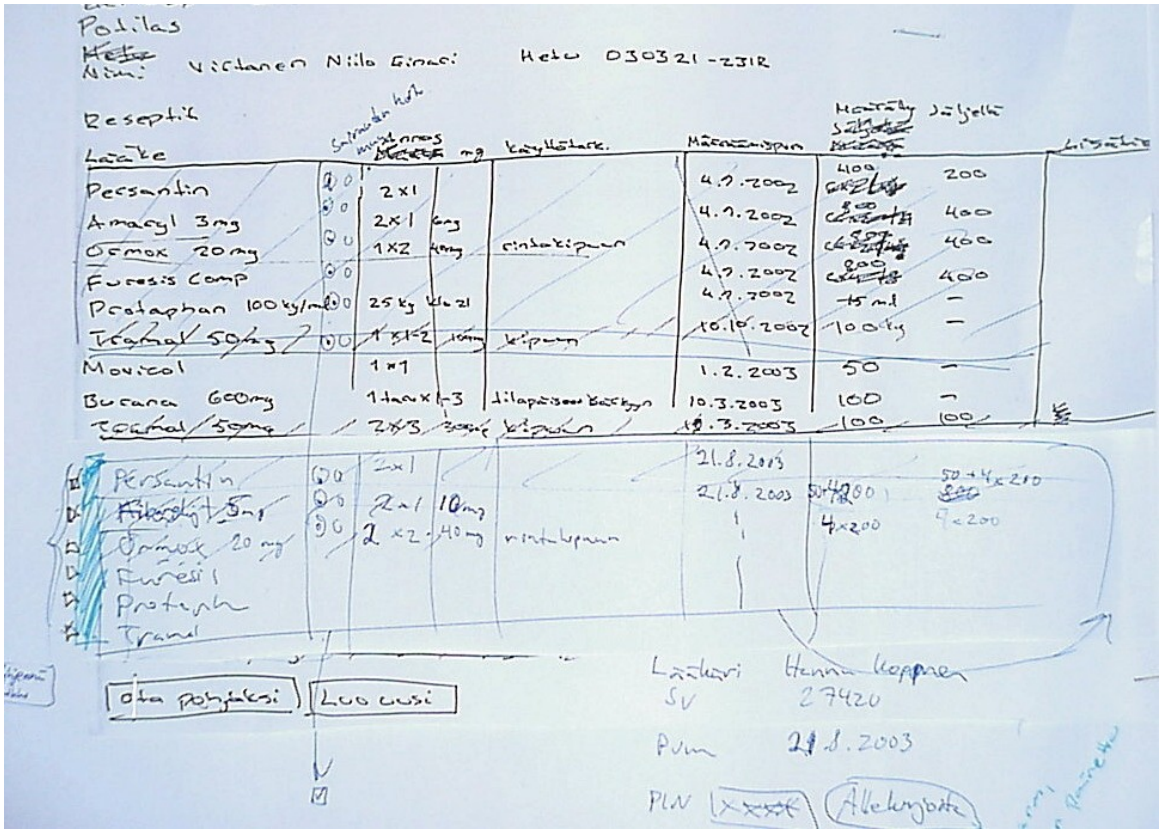
Toimenpide c. Refaktoroi käyttöliittymää tarvittaessa.

Kuva 3.6: GDD-suunnittelumenetelmän vaiheet [Laakso06a, s. 46].

Toteutusnäkökulmia ei oteta tässä vaiheessa suunnittelua huomioon, vaikka toteutuksessa käytettävät tekniikat ja työkalut olisi jo lyöty lukkoon ja ennalta tiedetty kankeiksi. Tarkoituksena on saada aikaan käyttäjän kannalta paras ratkaisu ja mahdolliset kompromissit tehdään myöhemmin tätä optimiratkaisua vasten. Jos käyttäjälle paras vaihtoehto ei ole tiedossa, on parhaan mahdollisen kompromissin suunnittelu vaikeaa. Lisäksi optimikäyttöliittymä saattaa olla käytettävyydeltään niin paljon parempi kuin kompromissiratkaisu, että mahdolliset toteutustekniikan vaihdosta aiheutuvat kustannukset kompensoituvat valmiin järjestelmän mahdollistamalla loppukäyttäjien tehokkaammalla työnteolla.

Kun ensimmäinen käyttötilanne voidaan suorittaa syntyneen käyttöliittymäratkaisun avulla, simuloidaan sen ilmeiset variaatiot (kuvan 3.6 vaihe 1.2). Variaatiot tarkoittavat muutoksia tilanteen yksityiskohdissa varsinaisen tilanteen pysyessä samana. Esimerkiksi jos verkkokirjakaupan suunnittelussa simuloidussa tilanteessa käyttäjä etsii ja ostaa ennalta tietämänsä Lonely Planet -matkaoppaan Espanjaan, voimme seuraavaksi simuloida variaation, jossa kirjaa ei ole saatavilla kaupassa ollenkaan. Kun ensimmäinen tilanne variaatioineen menee läpi, otetaan suunnitteluun seuraava käyttötilanne ja sen mahdolliset variaatiot (vaiheet 2.1 ja 2.2). Kirjakauppaesimerkissä seuraava käyttötilanne voi olla esimerkiksi sellainen, jossa käyttäjä etsii sopivaa matkaopasta Espanjaan ilman etukäteistietoa mahdollisista vaihtoehdoista. Toisessa käyttötilanteessa tarvittavat tiedot ja toiminnot integroidaan ensimmäisen tilanteen simuloinnin perusteella syntyneeseen käyttöliittymään. Kun toinenkin käyttötilanne voidaan suorittaa järjestelmällä, simuloidaan ensimmäinen tilanne käyttöliittymäratkaisulla uudestaan (vaihe 2.3). Näin varmistetaan, että toisen käyttötilanteen integroinnissa tehdyt muutokset eivät haittaa ensimmäisen tilanteen suorittamista. Sitten simulointiin otetaan mukaan kolmas tilanne ja tällä tavalla jatketaan edelleen.

Kuvassa 3.7 on esimerkki elektronisen reseptin järjestelmän suunnitteluvaiheesta muutaman käyttötilanteen simuloinnin jälkeen. Kuvaan on nyt määritelty yksi-käsitteisesti sellainen tietosisältö ja toimintalogiikka, jolla simuloinnissa mukana olleet käyttötilanteet pystyy suorittamaan järjestelmällä. Kukaan muu kuin suunnittelijat itse eivät kuitenkaan pysty päättelemään toimintalogiikkaa kuvasta,



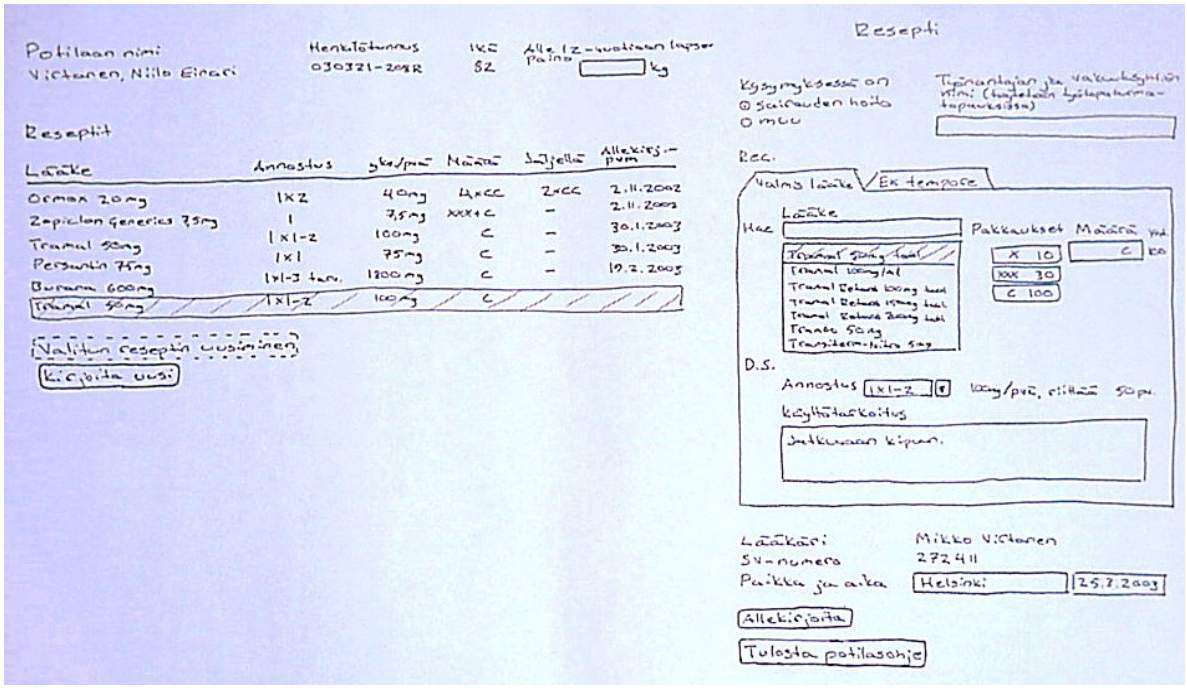
Kuva 3.7: Elektronisen reseptin käyttöliittymä muutaman käyttötilanteen simuloinnin jälkeen [Interacta03].

sillä ratkaisu on esitetty yksittäisen kuvan avulla. Määritelty toimintalogiikka on suurelta osin vain käyttöliittymän laatineiden suunnittelijoiden mielessä. Suunnitteluvaiheessa kuitenkin riittää, että suunnittelijat itse tietävät käyttöliittymän toimintalogiikan.

Kun kaikki käyttötilanteet on simuloitu, testataan käyttöliittymäratkaisua ketjuttamalla tilanteita peräkkäin (kuvan 3.6 toimenpide b). Tarkoituksena on simuloida siirtymät tilanteesta toiseen. Esimerkiksi elektronisen reseptin järjestelmää suunnitellessa käyttöliittymä saattaa jäädä reseptin uusimisen jälkeen sellaiseen tilaan, että uuden reseptin laatiminen heti perään ei ole mahdollista. Tämänkaltaiset aukot toimintalogiikassa vältetään ketjuttamalla tilanteita peräkkäin.

Kun kuvan 3.7 elektronisen reseptin käyttöliittymään integroitiin lisää tilanteita ja siihen generoitui lisää tietosisältöä ja toimintoja, huomattiin pian ettei yhteen suureen reseptitaulukkoon perustuva ratkaisu enää toiminut. Taulukkoon alkoi tulla niin

paljon kenttiä, että tarvittiin lisää tilaa esimerkiksi reseptin käyttötarkoituksen kirjaamista varten. Tällaisessa tilanteessa käyttöliittymää *refaktoroidaan* [Laakso05, s. 37-39]. Refaktoroinnissa käyttöliittymän rakenne jäsennetään uudelleen, mutta itse toimintoihin ja tietosisältöön ei kosketa. Kuvassa 3.8 näytetään refaktoroitu ja puhtaaksi piirretty näyttökuva, jossa kuvan 3.7 suuri reseptitaulukko on jäsennetty uudelleen *Overview beside Detail* -rakenteella [Laakso04b, s. 1-2]. Uuteen reseptitaulukkoon on jätetty vain lääkärin käyttötilanteiden kannalta kriittinen informaatio (*overview*), jonka perusteella hän paikantaa uusittavan reseptin ja näkee potilaan nykyisen lääkityksen. Taulukosta valittu resepti näytetään yksityiskohtaisesti näytön oikealla laidalla (*detail*).



Kuva 3.8: Puhtaaksi piirretty näyttökuva refaktoroinnin jälkeen [Interacta03].

Vaikka GDD-menetelmässä keskitytään erityisesti hyödyllisyyteen ja tehokkuuden optimointiin, ei opittavuus jää huomiotta. Kun käyttötilanteen suorittamisen polku on lyhyt, sekvenssissä on vähemmän kohtia potentiaalisille virheaskelille. Käyttäjän on helpompi muodostaa järjestelmän toiminnasta oikeanlainen mentaalimalli, kun kaikki tarvittavat tiedot ja toiminnot näytetään kerralla yhdessä ikkunassa usean sijaan [Lauesen01]. Jos tiedot ja toiminnot on jaoteltu useampaan ikkunaan, käyttäjä

joutuu navigoimaan eteenpäin nähdäkseen mitä järjestelmällä voi tehdä. Siten mentaalimalli muodostuu pala palalta käyttäjän navigoidessa järjestelmässä.

Simuloinnin tuloksena järjestelmään generoituu vain käyttötilanteissa tarvittavat tiedot ja toiminnot, eikä mitään muita. Näytön pinta-ala voidaan siis käyttää pelkästään käyttäjän tarvitsemiin toimintoihin ja dataan, eivätkä turhat toiminnot ja tiedot ole haittaamassa käyttötilanteiden suorittamista (vrt. luvussa 2.1 esitelty tutkimustulos: 45% toteutetuista toiminnallisuuksista oli sellaisia, joita kukaan käyttäjä ei koskaan tarvitse).

Opittavuudesta kannattaa joka tapauksessa huolehtia vasta kun tehokkuus on kunnossa. Tehokkuuden lisääminen opittavan käyttöliittymän päälle muuttaa tyypillisesti käyttöliittymäratkaisua niin paljon, että opittavuus pitää suunnitella alusta asti uudestaan [Laakso05, s. 39-41]. Tarkastellaan tilannetta, jossa opittavuus on pyritty optimoimaan ennen tehokkuutta. Oletetaan, että työtehtävän suorittaminen vaatii käyttäjältä kaksitoista ikkunasiirtymää käyttöliittymässä. Siirtymät ovat käyttäjälle itsestään selviä eli opittavuus on hyvä. Kun käyttöliittymän tehokkuutta ryhdytään parantamaan, kerätään kerralla tarvittavat tiedot ja toiminnot yhdelle näytölle ja turhat ikkunasiirtymät minimoidaan. Seurauksena osa siirtymistä häviää ja osa tehdään erilaisella käyttöliittymäratkaisulla, joka näyttää käyttäjälle enemmän tietoja ja toimintoja kerralla. Alkuperäiseen ratkaisuun suunniteltu opittavuus on hävinnyt ikkunasiirtymien mukana, mutta uudessa tehokkaammassa käyttöliittymä-ratkaisussa saattaa olla uusia erilaisia opittavuusongelmia. Nämä pitäisi vielä ratkaista.

Muiden käyttöliittymäsuunnittelumenetelmien tavat tuottaa käyttöliittymäratkaisu eroavat paljon GDD:n menettelyistä. Contextual Design keskittyy käyttäjän työn mallintamiseen erilaisilla tavoilla ennen paperiprototyyppien laatimista mallien perusteella [Beyer98, osat 2-6]. Menetelmästä puuttuu kuitenkin kuvassa 3.6 esitetyn kaltainen systemaattinen menettely käyttöliittymäratkaisun laatimiselle.

Cooperin Goal-Directed Design -menetelmässä laaditaan kenttätutkimusten perusteella persoonakuvauksia, jotka ovat kuvauksia kuvitteellisista kohderyhmään

kuuluvista loppukäyttäjistä [Cooper03, luvut 4-5]. Persoonat kuvailevat käyttäjän henkilökohtaisia ominaisuuksia ja kiinnostuksen kohteita sekä elämäntilannetta. Järjestelmän toimintalogiikkaa kiinnitetään kirjoittamalla tekstimuotoisia skenaarioita, joissa persoonan avulla kuvataan sekä käyttökontekstia järjestelmän ulkopuolella että persoonan ja järjestelmän välistä toimintaa [Cooper03, luku 6]. Skenaarioiden perusteella laaditaan korkean tason näyttökuvat, joiden yksityiskohtia tarkennetaan iteratiivisesti. Cooperin menetelmässä esimerkkihenkilöillä on keskeinen rooli, GDD:ssa puolestaan esimerkkitilanteilla. Laakson antamassa esimerkissä [Laakso06d] parturin ajanvarausjärjestelmän käyttöliittymän syntyyn eivät vaikuttaneet persoonien henkilökohtaiset ominaisuudet, kuten ”Niina on luonteeltaan eloisa ja vilkas, elämänmyönteinen ja rohkeakin”. Kriittistä käyttöliittymäsuunnittelulle olivat itse ajanvaraustilanteet, joissa käyttäjän oli järjestelmän avulla tehtävä päätös hiukset leikkaavasta parturista ja sopivasta leikkuuajasta omiin aikataulurajoitteihinsa nähden.

Cooperin menetelmän lisäksi monet muut menetelmät hyödyntävät skenaarioita käyttöliittymäsuunnittelussa [Go04]. Carrollin *skenaariopohjaisessa suunnittelussa (scenario-based design)* järjestelmän toimintalogiikkaa kiinnitetään kuvaamalla tekstimuotoisesti käyttäjän työnkulkua tulevassa järjestelmässä [Carroll94]. Työnkulun simulointi on yhteinen piirre GDD:n kanssa, mutta näyttökuvilla on etuja tekstikuvauksiin verrattuna. Yksityiskohtaisesti piirretyillä näyttökuvilla esitetty toimintalogiikka on yksikäsitteinen, mutta tekstiskenaarioissa kuvatut toimintalogiikat voivat olla ristiriidassa keskenään. Näyttökuvien avulla käyttöliittymää voidaan testata, skenaarioilla voidaan vain arvioida tekstillä kuvatun työnkulun suoraviivaisuutta.

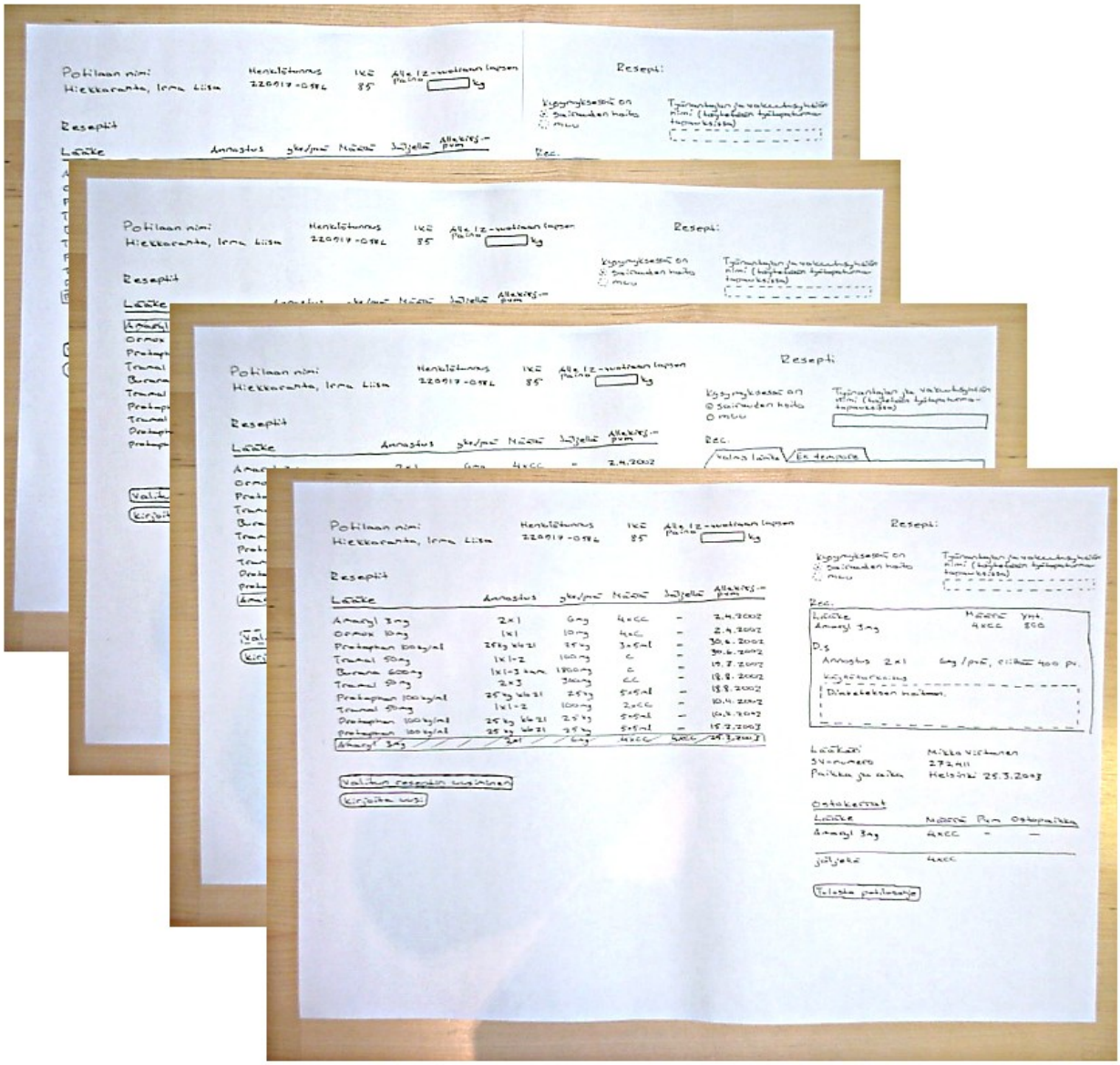
Muista menetelmistä Lauesenin Virtual Windows muistuttaa GDD-käyttöliittymäsuunnittelua eniten. Keskeisin GDD:n ja Virtual Windowsin yhteinen piirre on oikeiden työtehtävien simulointi. Virtual Windows -suunnittelussa virtuaali-ikkunat laaditaan simuloimalla selvitettyjä käyttäjien työtehtäviä eli *tehtäväkuvauksia (task descriptions)* [Lauesen05, luku 6]. Tehtäväkuvauksissa on sallittua kiinnittää järjestelmän toimintoja ja työnkulkuja tehtävän ratkaisemiseksi, mitä puolestaan pyritään välttämään GDD:n käyttötilanteissa. Toinen menetelmien keskeinen ero on

se, että virtuaali-ikkunoihin suunnitellaan vain tehtäväkuvauksissa tarvittava tietosisältö. Kaikki toiminnallisuus jätetään tässä vaiheessa pois, toiminnot lisätään vasta tietosisällön suunnitteluvaiheen jälkeen simuloimalla tehtäväkuvauksia uudelleen [Lauesen05, luku 7]. Virtuaali-ikkunoita läpikäydään loppukäyttäjien kanssa ja tietosisältöä korjataan läpikäyntien perusteella. Tässä on samoja piirteitä kuin GDD:n yhteydessä käytettävässä hyödyllisyysläpikäynnissä, jossa sekä tietosisältöä että toiminnallisuutta simuloidaan yhdessä loppukäyttäjien kanssa.

3.3.3 Käyttöliittymän dokumentointi

Toteutusta, testausta ja asiakasdemoja varten toimintalogiikka täytyy kommunikoida muille osapuolille. Tähän tarkoitukseen käyvät näyttökuvasarjat, joista käyttötilanteiden suorittamisen toimenpidesekvenssit selviävät. [Laakso06a, s. 57]. Kuvassa 3.9 on esimerkki elektronisen reseptin käyttöliittymää kuvaavasta kuvasarjasta. Käsin piirrettyyn kuvasarjaan on valmisteltu sekvenssi, jossa lääkäri uusii potilaan diabeteslääkityksen. Kuvasarjoja simuloitaessa nähdään, miten käyttötilanteet suoritetaan askel askeleelta: mistä käyttäjä painaa ja miten järjestelmä reagoi.

Toteutuksen syötteeksi toimintalogiikka kannattaa kuvata näyttökuvasarjoilla erityisesti silloin, jos järjestelmän toteutuksen tekee eri yritys tai jos käyttöliittymäsuunnittelijoilla ei ole mahdollista olla tiiviisti mukana toteutustyön tukena. Tarvittaessa näyttökuvat voidaan piirtää puhtaaksi tietokoneella, esimerkiksi käsin piirretyn käyttöliittymäratkaisun sovittamiseksi tiettyyn näyttöresoluutioon. Kuvaan 3.10 on tiivistetty kokonaisista tietokoneella puhtaaksi piirretyistä näyttökuvista koostuva elektronisen reseptin kuvasarja, jossa lääkäri uusii potilaan diabeteslääkereseptin. Lääkäri kirjoittaa potilaan henkilötunnuksen (näyttökuva nro. 1) ja potilaan aiemmat reseptit päivittyvät näkyville. Lääkäri valitsee taulukosta uusittavan reseptin (2), painaa ”Valitun reseptin uusiminen”-nappia (3), tarkistaa reseptin tiedot ja painaa ”Allekirjoita”-nappia (4). Toimintalogiikka on kuvattu merkkaamalla kuhunkin kuvaan käyttäjän suorittama toimenpide, jonka seuraus näkyy seuraavassa kuvassa. Näin kuvasarjoista käyvät ilmi tarvittavat toiminnot ja niiden interaktiotavat sekä järjestelmän tietosisältö käyttäjän kannalta.



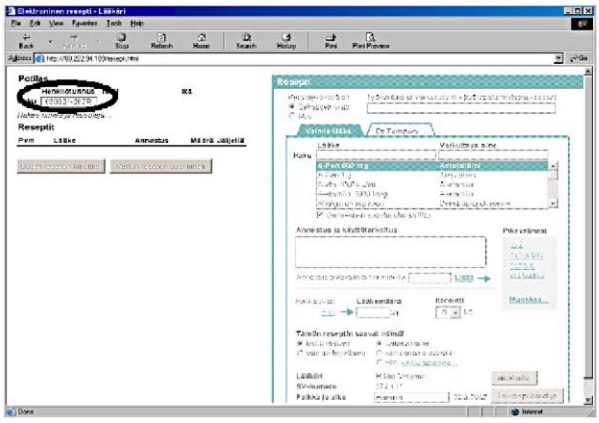
Kuva 3.9: Toimintalogiikka käsin piirretyllä näyttökuvasarjalla dokumentoituna [Interacta03].

Pelkkään toimintalogiikan demonstroimiseen ja käyttöliittymän testaamiseen kelpaavat myös alkeellisemmat prototyypit kuin täydelliset kuvasarjat. Paperi-prototyyppiä voi päivittää esimerkiksi paperilapuilla tai kynällä piirtämällä demon aikana [Laakso06a, s. 64-66]. Tämä ei ole niin työlästä kuin kuvasarjojen laatiminen.

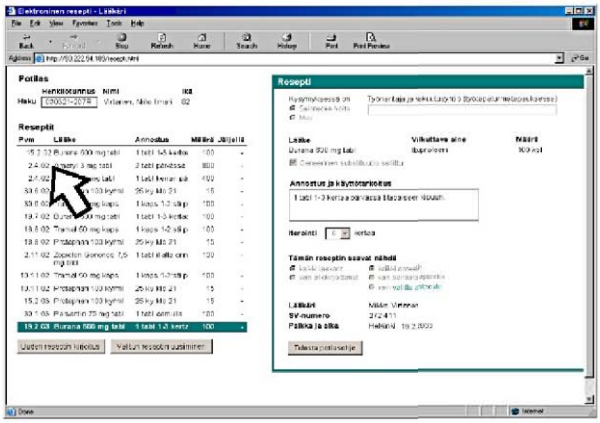
Alkeellisempi prototyyppi riittää toteutustyön syötteeksi, jos käyttöliittymäsuunnittelija on aktiivisesti mukana demonstroimassa toimintalogiikkaa toteutuksen aikana. Kuvasarjojen laatimisen sijaan käyttöliittymäsuunnittelija demonstroi

toimintalogiikan suoraan toteutustiimille ja on tarpeen mukaan käytettävissä tarkennuksia varten.

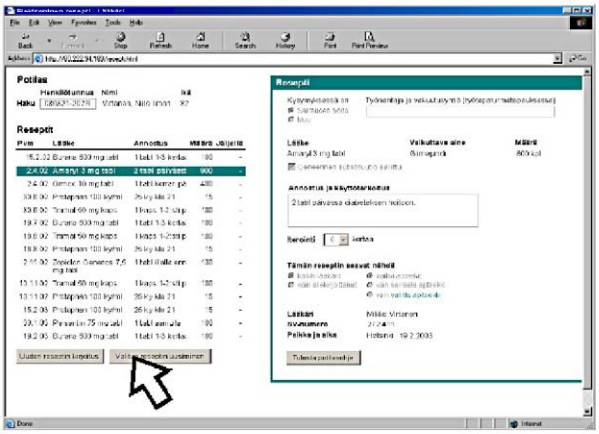
1



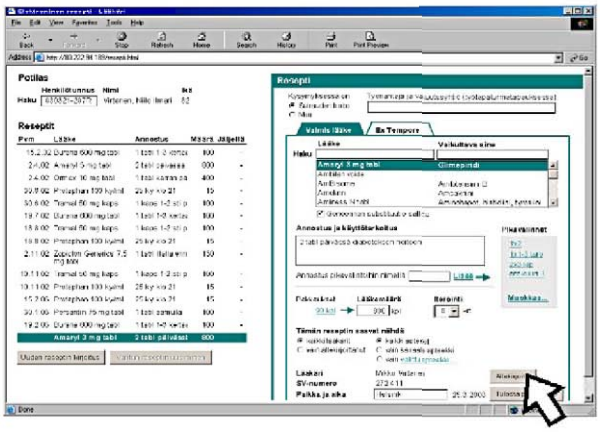
2



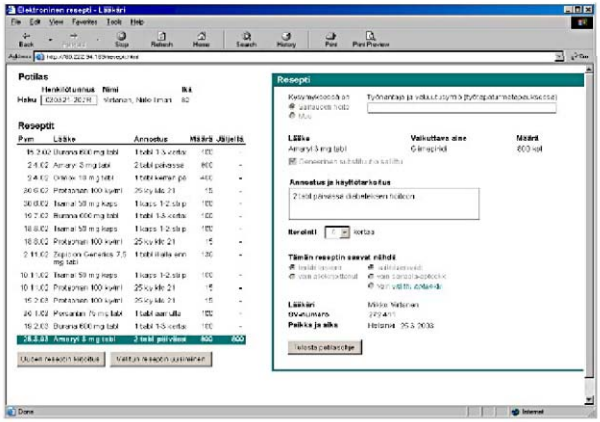
3



4



5



Kuva 3.10: Reseptin uusimisen toimintalogiikka tietokoneella puhtaaksi piirretyn kuvasarjan avulla dokumentoituna [Interacta03].

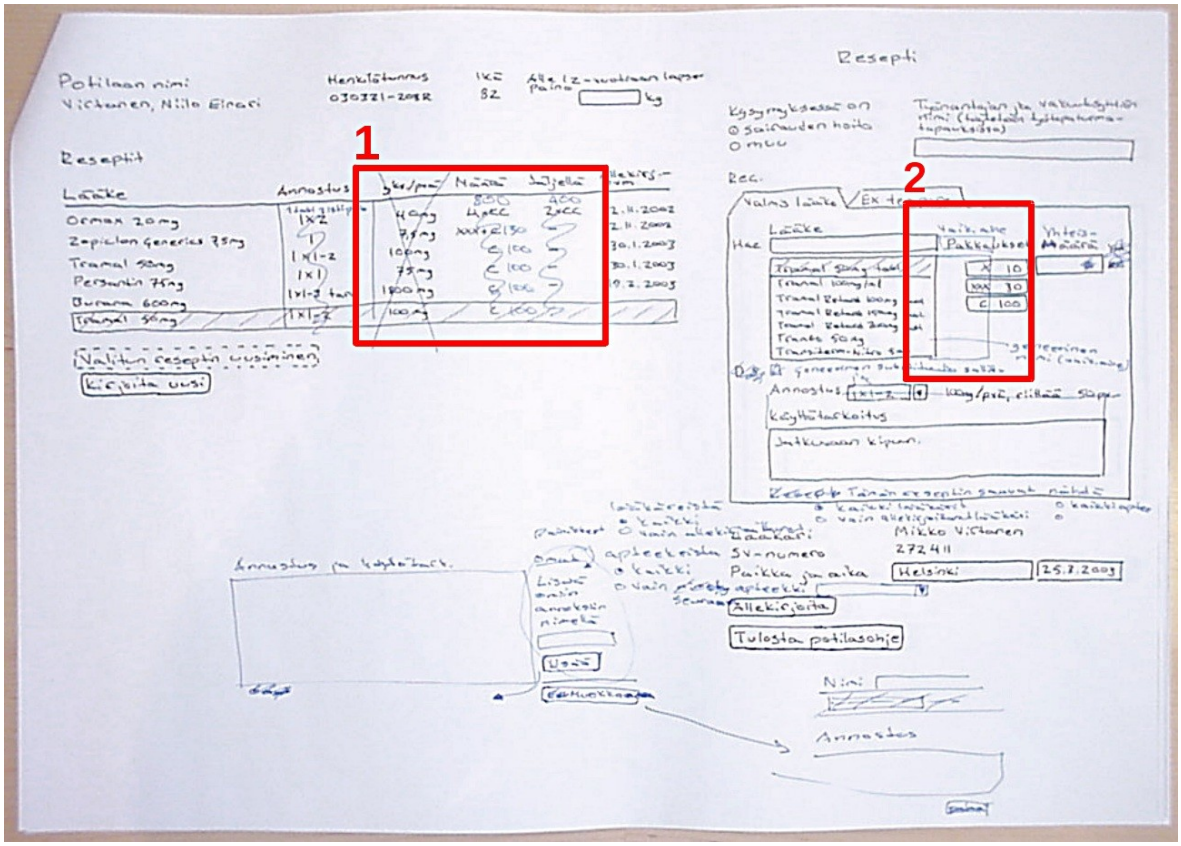
3.3.4 Käyttöliittymäprototyyppien testaus ja arviointi

Suunnitteluvaiheessa olevan käyttöliittymän arviointiin kannattaa käyttää *hyödyllisyysläpikäyntiä (utility walkthrough)* [Laakso06c, s. 18-24]. Hyödyllisyys-läpikäynnin tavoitteena on selvittää suunnitteilla olevan käyttöliittymän tietosisällön ja toiminnallisuuden puutteita ja ongelmia. Lisäksi pyritään saamaan kiinni mahdollisia suunnittelusta puuttuvia käyttötilanteita. Opittavuuteen ei kiinnitetä tässä vaiheessa huomiota.

Kuvan 3.9 käsin piirretyistä näyttökuvista koostuva prototyyppi oli laadittu käyttöliittymän ensimmäisestä versiosta hyödyllisyysläpikäyntiä varten. Kuvasarjaan oli dokumentoitu lääkärin työnkulku, jossa potilaan resepti uusitaan. Paperi-prototyypin avulla lääkärin suunniteltua työnkulkua voidaan simuloida hänen kanssaan yksityiskohtaisesti askel askeleelta. Tarkoituksena on löytää prototyypistä sellaisia kohtia, joissa käyttäjän ongelmanratkaisustrategia poikkeaa käyttöliittymään suunnitellusta logiikasta. Tarvittaessa prototyypin toimintaa voidaan muuttaa lennossa leikkelemällä sitä ja piirtämällä puuttuvaa toiminnallisuutta ja tietosisältöä näkyviin. Tämä on kuitenkin vain keino ymmärtää käyttäjän päätöksentekoa, eikä tarkoituksena ole kiinnittää lopullisia suunnitteluratkaisuja. Palaverin jälkeen arvioidaan, mitä muutoksia kannattaa oikeasti tehdä.

Esimerkkejä hyödyllisyysläpikäynnin tuloksista näytetään kuvassa 3.11. Laatikolla 1 merkatussa kohdassa suunnittelija on vetänyt rastilla yli lääkkeen vaikuttavan aineen päiväkohtaisen yhteismäärän, joka oli tarpeetonta näyttää reseptitaulukossa. Lisäksi suunnittelija on vaihtanut määrien ilmaisun roomalaisista numeroista arabialaisiksi, sillä paperiresepteissä käytetyt roomalaiset numerot osoittautuivat vanhaksi jäänteeksi ilman erityistä syytä. Merkatussa kohdassa 2 suunnittelija on piirtänyt taulukkoon hakukentän ja uuden sarakkeen, jotka mahdollistavat lääkkeen haun sen nimen lisäksi vaikuttavalla aineella. Arvioitava käyttöliittymä ei tukenut tätä tarvetta.

Esille nousi myös suunnittelusta puuttuvia käyttötilanteita. Lääkärillä oli sellaisia potilaita, jotka hakivat lääkkeet arkaluontoiseen sairauteensa tavallisen apteekin sijaan sairaala-apteekista. Tavalliset apteekit eivät saaneet nähdä potilaan reseptejä,



Kuva 3.11: Hyödyllisyysläpikäynnissä ilmi tulleita puutteellisia kohtia käyttöliittymässä [Interacta03].

eikä käyttöliittymä tukenut vielä tällaisia tilanteita. Näin hyödyllisyysläpikäynneillä saadaan ongelmakohtien paikantamisen lisäksi täydennettyä kenttätutkimusaineistoa.

Contextual Design -menetelmän *prototyyppihaastatteluissa (prototype interview)* [Beyer98, s. 396-411] on paljon samoja elementtejä kuin hyödyllisyysläpikäynnissä. Myös prototyyppihaastatteluissa simuloidaan käyttäjän oikeita työtehtäviä ja prototyypin avulla pyritään ymmärtämään käyttäjän työtä paremmin. Keskeinen ero hyödyllisyysläpikäyntiin on se, että käyttäjiä pyydetään korjaamaan prototyyppiä itse ja heiltä kysytään mielipiteitä suunnittelijoiden laatimista erilaisista ratkaisuista. Nämä ovat ongelmallisia menettelyjä. On käyttöliittymäsuunnittelijan tehtävä suunnitella ratkaisut ongelmiin, sillä käyttäjät eivät tyypillisesti osaa käyttöliittymäsuunnittelua [Cooper03, s.9]. Käyttäjien mielipiteet eivät ole luotettava lähde käyttöliittymien arviointiin, vaan erilaisten ratkaisujen paremmuus selviää suorittamalla niillä oikeita työtehtäviä [Nielsen01].

Käyttöliittymäratkaisun hyödyllisyyttä ja tehokkuutta voidaan testata myös ilman loppukäyttäjiä. *Simulointitestauksessa* arvioija ottaa yhden konkreettisen käyttötilanteen kerrallaan ja simuloi käyttöliittymän tarjoamaa tilanteen optimaalista suorituspolkua läpi useaan kertaan [Laakso06c, s. 2-17]. Tästä optimaalisesta polusta arvioija paikantaa sekä hyödyllisyys- että käytettävyysongelmia. Tehokkuusongelmien ja puuttuvan toiminnallisuuden ja tietosisällön lisäksi kiinni jää myös pahimpia opittavuusongelmia. Hyödyllisyysläpikäyntiä ei kuitenkaan voi korvata simulointitestauksella, sillä simulointitestaajalla ei ole vastaavaa tietämystä työtehtävistä kuin läpikäyntiin osallistuvilla ammattinsa asiantuntijoilla. GDD-menetelmällä suunnitellussa järjestelmässä käyttötilanteet on simuloitu suunnittelun aikana läpi useaan kertaan ja käyttöliittymään on pyritty tuottamaan niille tehokkuuden kannalta optimaalinen toiminnallisuus, tietosisältö ja interaktiotavat. On silti järkevää, että toinen käyttöliittymäasiantuntija simulointitestaa paperiprototyypit.

Kun hyödyllisyys ja tehokkuus on saatu optimoitua, kannattaa opittavuutta testata erityisesti järjestelmän tullessa suuremman yleisön käyttöön. Ilman loppukäyttäjiä opittavuusongelmia voidaan löytää *kognitiivisella läpikäynnillä (cognitive walkthrough)* [Lewis97]. Kognitiivisessa läpikäynnissä arvioija seuraa käyttötilanteen oikeaa suorituspolkua samaan tapaan kuin simulointitestauksessa. Jokaisen askeleen kohdalla vastataan neljään tarkistuskysymykseen. Kysymysten avulla testauksen tuloksena saadaan suorituspolussa vastaan tulevia opittavuusongelmia: kohtia käyttöliittymässä, jotka ovat vaikeita keksiä tai houkuttavat väärän toiminnon käyttöön.

Loppukäyttäjien kanssa opittavuutta voidaan arvioida *käytettävyysläpikäynneillä (usability walkthrough)* [Bias91] tai *käytettävyystesteillä (usability testing)* [Nielsen93, luku 6]. Käytettävyysläpikäyntiä varten kullekin osallistuvalle käyttäjälle valmistellaan kuvasarja käyttötilanteen oikeasta suorituspolusta. Kuvasarjaa askel askeleelta läpi käydessä käyttäjiä pyydetään valitsemaan näyttökuvasta se toimenpide, jonka he tekisivät seuraavaksi. Käyttäjien tekemien valintojen syistä keskustellaan ja sen jälkeen sekvenssiä jatketaan oikeaa suorituspolkua edeten. Näin saadaan selville kohdat, joissa käyttäjät olisivat valinneet väärän toiminnon ja perustelut virheellisille valinnoille.

Käytettävyystestissä käyttäjälle annetaan testitehtävinä käyttötilanteita, jotka käyttäjä suorittaa itsenäisesti käyttöliittymän avulla. Käyttäjä voi itse valita strategian tehtävän suorittamiseen ja käyttää prototyypin toimintoja parhaaksi katsomallaan tavalla, joten prototyypin täytyy olla huomattavasti paremmin valmisteltu kuin käytettävyysläpikäynnissä. Käytettävyystestin resursseilla voidaan tehdä useita käytettävyysläpikäyntejä, joten useampi testaus-korjaus -iteraatio läpikäynneillä on harkitsemisen arvoinen vaihtoehto. Kognitiivinen läpikäynti vaatii vielä vähemmän resursseja, mutta kirjallisuuden mukaan sillä löydetään vain n. 40% käytettävyys-testeillä löydettävistä ongelmista [Lewis97].

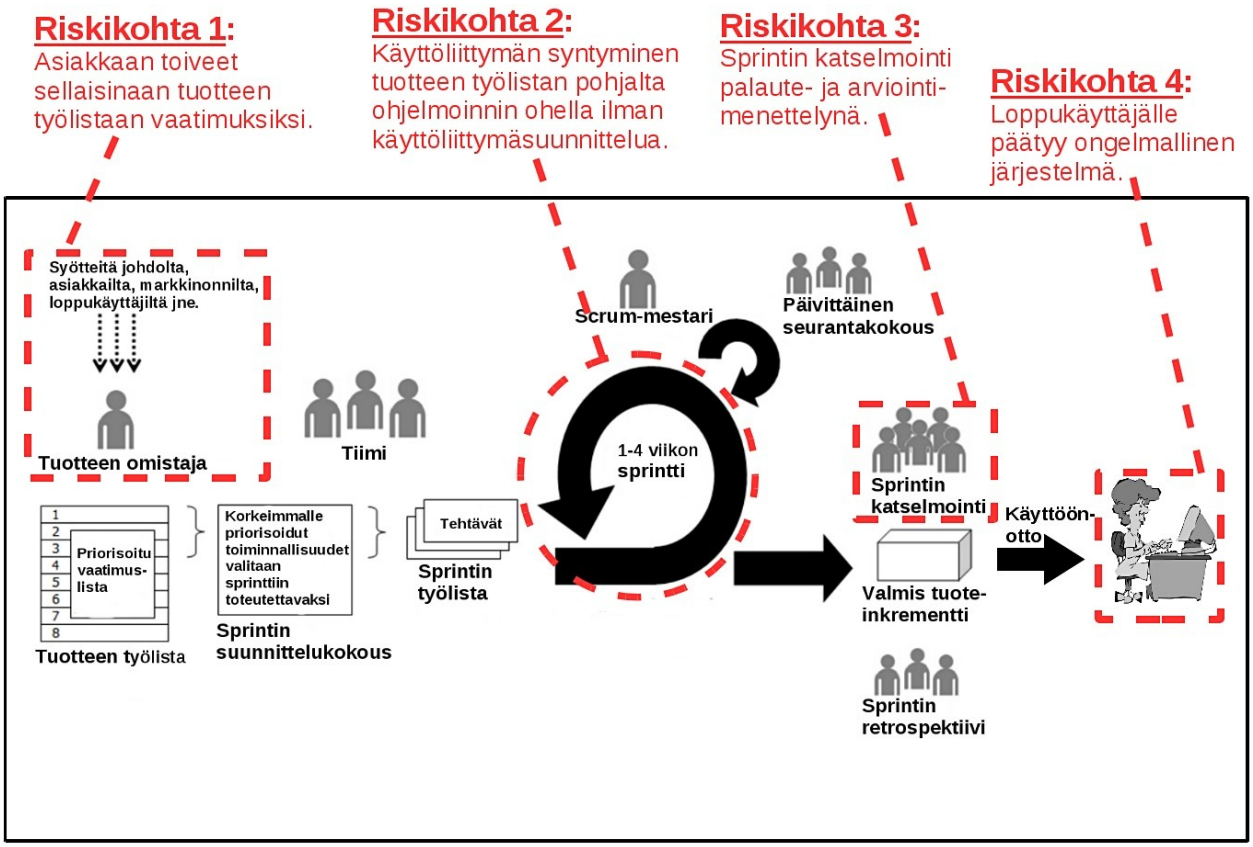
Resursseja ei kannata käyttää tietokoneella toimivan prototyypin laatimiseen testausta varten, sillä paperiprototyypeillä löydetään samat oleelliset ongelmat kuin toimivilla tietokoneprototyypeillä [Virzi96]. Paperiprototyyppien laatiminen ja muokkaaminen on huomattavasti nopeampaa kuin tietokoneella toimivien proto-tyyppien ja yhden toimivalla prototyypillä tehdyn testin resursseilla voidaan tehdä useita testejä paperiprototyypeillä [Rettig94].

4 Käyttöliittymäriskit Scrum-prosessimallissa

Ohjelmiston käyttöliittymä on yksi ohjelmistotuotannon keskeisiä riskikohtia. Järjestelmään suunnitellaan ja toteutetaan usein puutteellinen tai väärä toiminnallisuus ja tietosisältö tai työtehtävien kannalta epäoptimaalinen käyttöliittymä. Tässä luvussa tarkastellaan, kuinka suuria edellä mainitut käyttöliittymäriskit ovat Scrum-prosessimallissa.

Kuvassa 4.1 näytetään Scrum-sprintin kulku lisättynä käyttöönottovaiheella. Käyttöönottoja voi olla useita projektin aikana, sillä kaikki tuoteinkrementit ovat tuotantovalmista koodia. Kuvasta on korostettu neljä riskikohtaa käyttöliittymä-suunnittelun näkökulmasta. Riskikohta 1 on tuotteen työlistan syntyminen. Listan sisältö voi tulla mistä vain, tyypillisesti se sisältää asiakkaan toivomia ominaisuuksia. Riskikohta 2 on käyttöliittymän syntyminen työlistan pohjalta ohjelmoinnin ohella ilman käyttöliittymäsuunnittelua. Riskikohta 3 on sprintin katselmoinnin käyttäminen palautteen saamiseen ja toteutetun tuoteinkrementin arviointiin. Riskikohtia 1-3 käsitellään vastaavissa aliluvuissa 4.1-4.3.

Riskikohta 4 on itse asiassa kolmen edellä mainitun seuraus: loppukäyttäjälle päätynyt ohjelmisto on hyödyllisyyden tai käytettävyyden osalta ongelmallinen. Tämä riski ei ole Scrum-menetelmän erityispiirre, vaan se koskee kaikkia prosessi-malleja. Jokaisen projektin tuottama ratkaisu verifioidaan joka tapauksessa oikeaa käyttöä vasten loppukäyttäjien ryhtyessä suorittamaan järjestelmällä tehtäviään. Tässä luvussa analysoitavat riskikohdat 1-3 lisäävät riskikohdan 4 toteutumisen todennäköisyyttä.



Kuva 4.1: Käyttöliittymäriskit Scrum-prosessimallissa (pohjaksi otettu kuva lähteestä [Sutherland07]).

4.1 Riskikohta 1: asiakkaan toiveet vaatimuksiksi

Scrum-menetelmässä tekeillä olevan ohjelmiston lähtökohtana on tuotteen työlista. Työlistaan kootaan usein asiakkaan edustajien toiveita tulevasta järjestelmästä ja se sisältää ”kaiken mitä kuka tahansa ajattelee tuotteessa tarvittavan tai olevan hyvä idea tuotteeseen” [Schwaber01, s. 33]. Asiakkaan pyytämien ominaisuuksien toteuttaminen sellaisenaan sisältää riskejä. Tyypillisesti asiakkaat pystyvät kyllä kertomaan mitä he haluavat, mutta tämä ei ole sama asia kuin se mitä he oikeasti tarvitsisivat. Asiakas ei usein tiedä sitä, mitä hän voisi saada [Orr04]. Esimerkiksi ammattikorkeakoulun opetuksen suunnittelujärjestelmässä asiakas oli pyytänyt raportointiominaisuutta, jossa käyttäjä voisi hakea opetustapahtumia millä tahansa kentillä [Laakso07]. Todellinen ongelma selvitettiin käyttäjähaastatteluilla ja osoittautui, että kyseinen haku palvelisi kahta konkreettista käyttötilannetta: opettaja kirjoittaa omista kursseistaan sisältökuvauksen ja aikataulun opiskelijoille, ja hallinnon työntekijä tekee tilastokeskukselle raportin edellisen vuoden opetuksesta.

Opettaja tarvitsi näytölle kerralla pelkät omat kurssinsa ja hallinnon työntekijä tilastokeskuksen tarvitsemat tiedot. Jos raportointiominaisuus olisi toteutettu, olisivat loppukäyttäjät joutuneet joka kerta syöttämään hakukriteerit ja käynnistämään haun saadakseen edellä mainitut tiedot näkyville. Raportti-ominaisuuden sijaan opettajalle kannatti näyttää suoraan lista omista kursseista ja hallinnon työntekijälle antaa valmiiksi tilastokeskuksen tarvitsema raporttimuoto. Hakua millä tahansa kentillä ei tarvittu. Toistuvista hakukriteerien syötöistä ja hakujen käynnistämisistä olisi seurannut turhia toimenpiteitä loppukäyttäjille eli tehokkuusongelma. Siten asiakkaan toivoman raportointiominaisuuden toteuttaminen olisi johtanut huonompaan käytettävyyteen.

Järjestelmästä voi tulla ongelmallinen myös hyödyllisyyden kannalta, eli siinä on joko puutteellinen tai väärä toiminnallisuus ja tietosisältö. Seuraava esimerkki on laadittu Reaktorin käyttöliittymäsuunnittelijan Karri-Pekka Laakson haastattelun pohjalta: Eräässä projektissa Reaktor toteutti järjestelmää teleoperaattorin tarpeisiin. Teleoperaattoria edustava tuotteen omistaja ilmoitti Reaktorin Scrum-tiimille, että heidän asiakkaanaan toimivalle sähköntoimittajalle tarvitaan järjestelmä. Sähkön-toimittaja tilasi teleoperaattorilta dataliittymiä etäluettaviin sähkömittareihinsa. Perinteistä Scrum-prosessia käyttäen projekti olisi voinut jatkua siten, että tuotteen omistaja listaisi tarvittavia ominaisuuksia tuotteen työlistaan, kuten toiminnot *liittymien tilaus, liittymien sulkeminen* ja *liittymien muokkaaminen.* Tiimi toteuttaisi järjestelmän tuotteen omistajan antamien vaatimusten perusteella ja lopulta ohjelmisto otettaisiin käyttöön. Jos järjestelmä olisi toteutettu edellä kuvatulla tavalla, olisi ollut olemassa riski, että syntyneestä järjestelmästä jäisi pois tärkeä sähköntoimittajan käyttötilanteissa tarvittava tieto: sähkömittarin numero.

Mittarinumeron puuttuminen aiheuttaa ongelmia esimerkiksi sähkömittarin rikkoutuessa, jolloin mittarin liittymä täytyy sulkea: Sähköntoimittaja joutuu avaamaan mittarin fyysisesti, irrottamaan SIM-kortin ja etsimään suljettavan liittymän järjestelmästä SIM-kortin numeron perusteella. Kun järjestelmä mahdollistaa liittymän haun sähkömittarin numeron perusteella, ei mittaria tarvitse avata fyysisesti eikä SIM-korttia kaivaa esiin. Tuotteen omistaja ei ollut tietoinen mittarinumeron tarpeesta, vaan se huomattiin Reaktorin selvittäessä sähkön-

toimittajan käyttötilanteita käyttöliittymäsuunnittelun syötteeksi. Lisäksi selvisi, että kuhunkin mittariin kytketystä liittymästä pidettiin kirjaa Excel-taulukossa, eli rikkoutunut mittari ja suljettu liittymä täytyy päivittää myös Excel-taulukkoon. Jos uusi järjestelmä sisältäisi tarvittavat tiedot mittareista ja liittymistä, voitaisiin Excel-taulukon manuaalisesta ylläpidosta päästä eroon.

Tuotteen työlistan toiminnot ovat ratkaisuja asiakkaan ongelmiin, mutta Scrum-menetelmässä itse ratkaistavia ongelmia ei välttämättä kuvata missään. Pyydettyjen ominaisuuksien toteuttaminen ilman varsinaisen ongelman analysointia johtaa helposti käyttäjän kannalta ongelmalliseen järjestelmään. Riski koskee sekä projektin alussa tuotteen työlistaan kirjattuja toiminnallisuuksia että toteutussprinttien aikana tulevia uusia toiminto- ja muutospyyntöjä.

Scrum-menetelmässä vastuu ongelmallisesta määrittelystä jää yleensä asiakkaalle, joka joutuu itse laatimaan ratkaisuja ongelmiinsa [Laakso07]. Parempaan lopputulokseen päästään, kun ohjelmiston toimittaja auttaa selvittämään todellisen ongelman ja laatimaan parhaan ratkaisun siihen [Orr04, Robertson02].

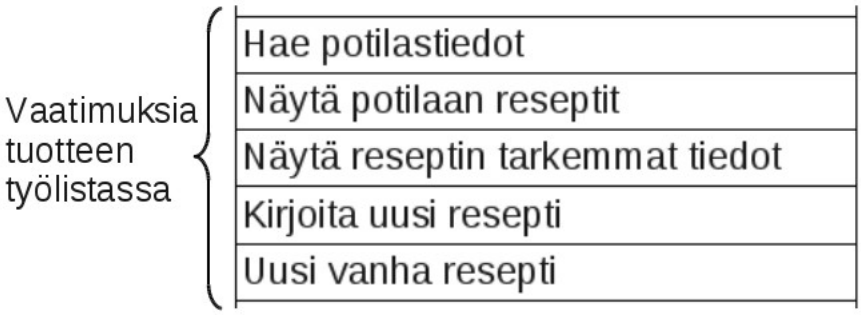
4.2 Riskikohta 2: käyttöliittymän syntyminen

Scrumin toteutussprintissä käyttöliittymäratkaisun laatii ohjelmoija, ei käyttöliittymäsuunnittelija. Tämä on ongelmallista, sillä ohjelmoijilla ei yleensä ole osaamista käyttöliittymäsuunnittelusta [Martin06]. Seurauksena on helposti järjestelmä, joka tukee käyttäjän työtehtäviä huonosti [Bailey93, Martin06]. Tyypillinen ohjelmoijan suunnitteluvirhe on laatia erillinen ikkuna jokaista järjestelmän toimintoa varten, jolloin tuloksena on paha tehokkuusongelma käyttäjän kannalta [Cooper03, s. 25]. Vaikka ohjelmoijilla olisikin käyttöliittymä-suunnitteluosaamista, ei ratkaisuja kannata laatia koodaustyön yhteydessä, koska tällöin ohjelmoijat tekevät väistämättä suunnittelupäätöksiä toteutuksen vaikeuden perusteella [Cooper95, s. 547]. Hyvät käyttöliittymäratkaisut ovat tyypillisesti vaikeampia toteuttaa kuin huonot, joten helpompia toteutusvaihtoehtoja suosimalla ei päädytä käyttäjän kannalta optimaaliseen ratkaisuun. On ylipäätään kohtuutonta vaatia, että ohjelmoijat pystyisivät haastavan koodaustyön ohella suunnittelemaan

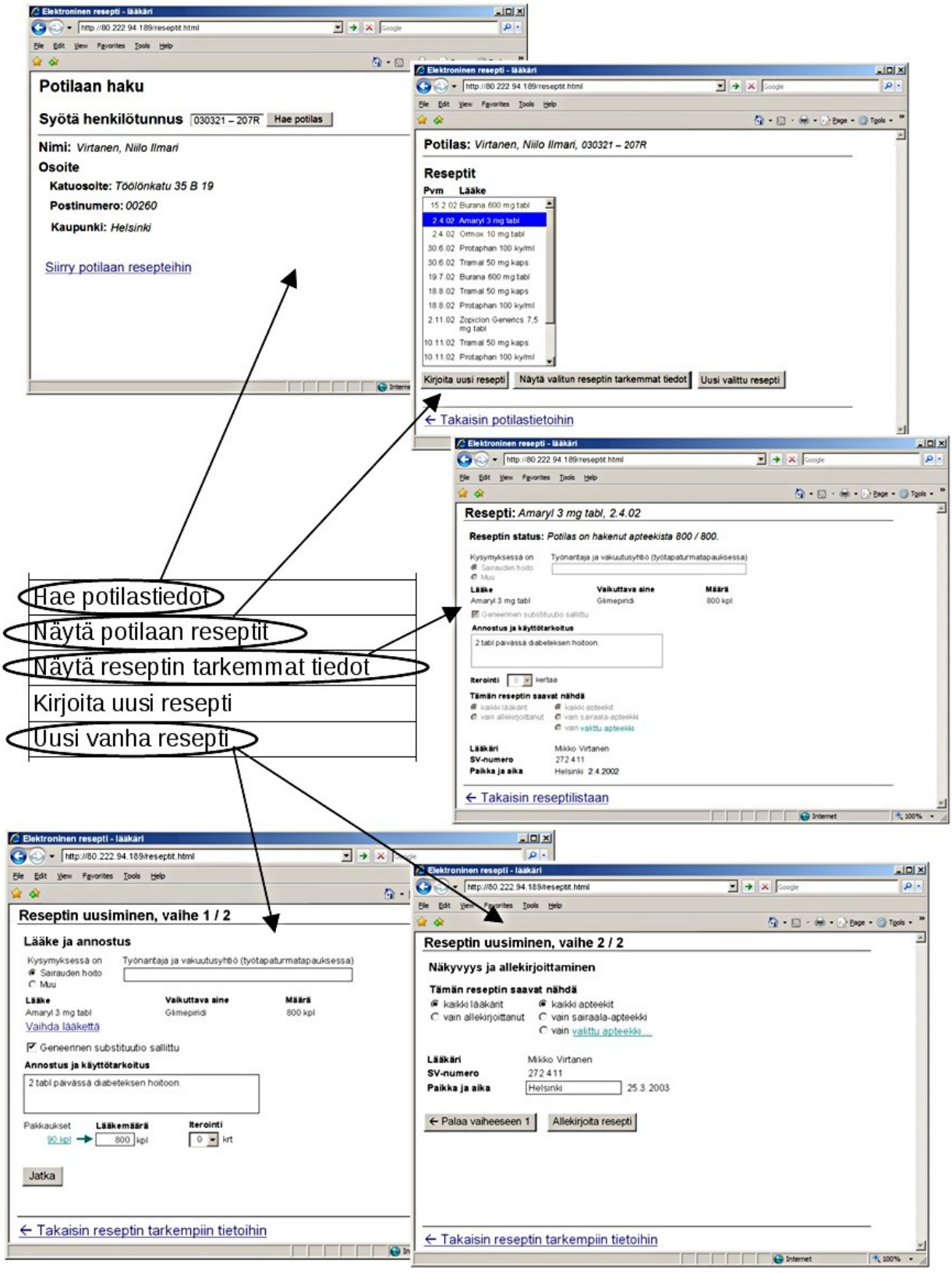
loppukäyttäjän työtehtäviä suoraviivaisesti tukevan käyttöliittymän [Cooper95, s. 547].

Käyttöliittymän laatiminen tuotteen työlistan toimintojen perusteella on riskialtis menettely. Toimintolistojen pohjalta suunnitellut järjestelmät paljastuvat usein käytön kannalta epäoptimaalisiksi, kun niitä simuloidaan oikeilla käyttötilanteilla [Laakso06, s. 58-60]. Seuraava esimerkki havainnollistaa toimintolistan pohjalta toteuttamisen ongelmia.

Oletetaan, että elektronisen reseptin järjestelmää ollaan toteuttamassa Scrum-menetelmällä. Sprinttiin on valittu toteutettavaksi kuvassa 4.2 näytetty osa tuotteen työlistasta. Lista on luotu tätä esimerkkiä varten kuvan 3.10 toiminnallisuuden perusteella ja se sisältää lääkärin tarvitseman toiminnallisuuden reseptin uusimista varten. Riskiä virheelliselle työlistalle ei ole, koska kuvan 3.10 käyttöliittymäratkaisu on suunniteltu GDD-menetelmällä simuloimalla ja se on testattu hyödyllisyys-läpikäynneillä. Etukäteen on siis selvillä, että sprinttiin valitut toiminnot ovat lääkärin käyttötilanteille keskeisiä ja lääkärin työnkulkua hyvin tukeva käyttöliittymäratkaisu sisältää juuri ne. Tiimi laatii sprintin työlistan parhaaksi katsomallaan tavalla ja toteuttaa toiminnot sprintin aikana. Toimintolistan pohjalta toteuttaminen johtaa helposti kuvan 4.3 tilanteeseen: jokainen tuotteen työlistan toiminto on päätynyt yhteen tai kahteen erilliseen ikkunaan.



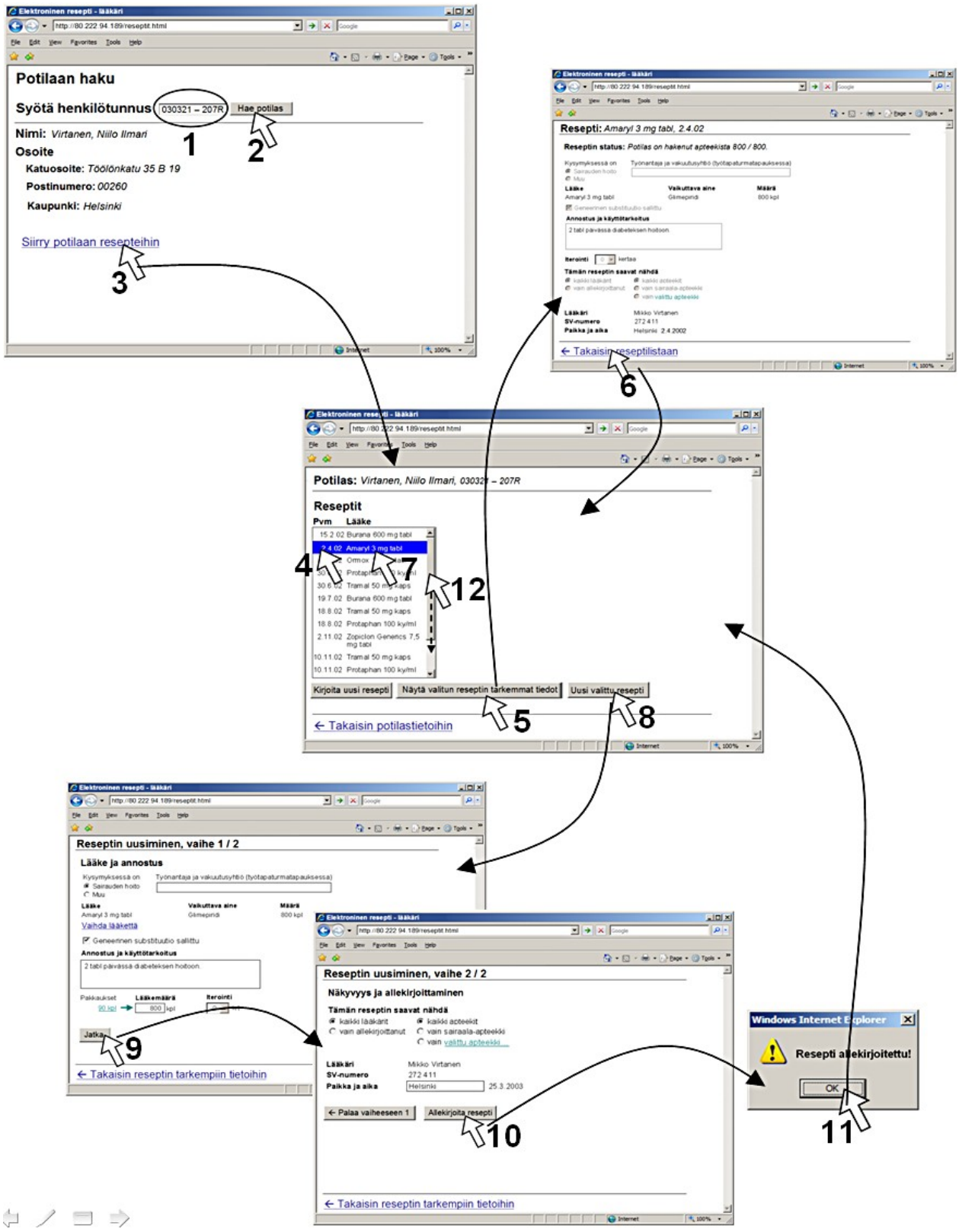
Kuva 4.2: Esimerkkiä varten laadittu elektronisen reseptin järjestelmän tuotteen työlistan osa.



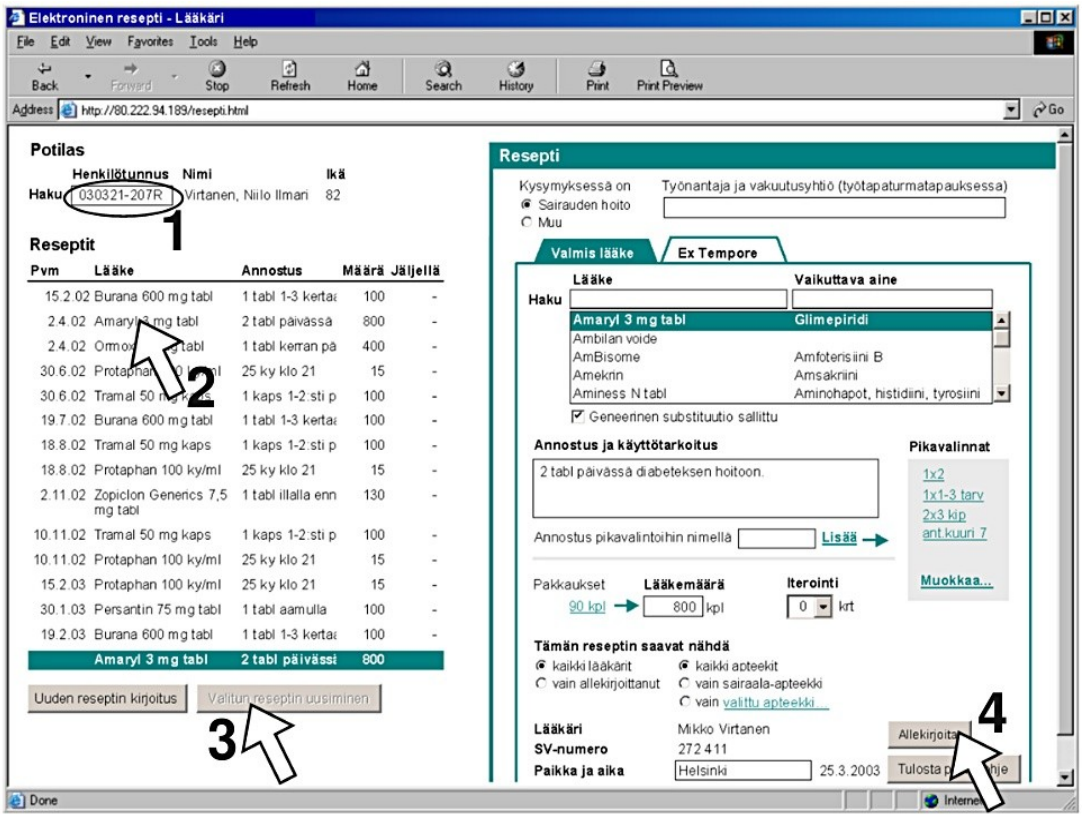
Kuva 4.3: Toiminnot erillisissä ikkunoissaan.

Käyttöliittymäratkaisun ongelmallisuus alkaa näkyä, kun sitä simuloidaan oikealla käyttötilanteella. Kuvassa 4.4 näytetään toimenpiteet ja ikkunasiirtymät, jotka lääkäri suorittaa uusiessaan potilaan diabeteslääkereseptin. Ensin lääkäri hakee potilaan reseptit näkyville (vaiheet 1-3). Lääkäri valitse uusittavan reseptin ja tarkistaa, onko potilaalla hakematta reseptin lääkkeitä apteekista (vaiheet 4-6). Lääkäri valitsee reseptin uusintaa varten, tarkistaa reseptin tiedot, uusii reseptin ja tarkistaa että resepti on nyt uusittu (vaiheet 7-12). Käyttöliittymäratkaisu vaatii lääkäriltä seitsemän ikkunasiirtymää ja kaksitoista toimenpidettä. Kuvassa 4.5 näytetään vastaavat ikkunasiirtymät ja toimenpiteet GDD-menetelmällä simuloimalla syntyneestä käyttöliittymäratkaisusta (kuvan 3.10 käyttöliittymäratkaisu). Ikkuna-siirtymiä on nolla ja toimenpiteitä neljä. Toimintolistan pohjalta suunnitellussa käyttöliittymässä on huonompi käytettävyys, sillä lääkäri joutuu suorittamaan monivaiheisen sekvenssin jatkuvasti useita kertoja työvuoron aikana. Tämä on selvä tehokkuusongelma.

Søren Lauesen kutsuu edellä kuvattua ongelmallista suunnittelutapaa *tehtäväpohjaiseksi suunnitteluksi (task-driven design)* [Lauesen05, s. 212-215]. Reseptilistasivulla (kuvan 4.4 keskimmäinen näyttö) näytetään tehtäväpainikkeet, joista kukin johtaa omaan ikkunaansa tai ikkunaketjuunsa. Edellä kuvatun työ-tehtävän tehokkuusongelman lisäksi Lauesen esittää toisenkin riskin. Kun ohjelmoijat laativat ikkunoita eri tehtäville, syntyneet ikkunat ovat usein eri näköisiä. Samat tiedot tai toiminnot voivat päätyä järjestelmään useammalla eri tavalla esitettynä, jolloin esimerkiksi resepti voi olla rakenteeltaan tai ulkonäöltään erilainen reseptiä uusittaessa kuin uutta reseptiä kirjoittaessa.



Kuva 4.4: Käyttöliittymä paljastuu simuloinnissa ongelmalliseksi.



Kuva 4.5: GDD-menetelmällä suunniteltu ratkaisu (näyttökuva lähteestä [Interacta03]).

4.3 Riskikohta 3: sprintin katselmointi

Sprintin katselmoinnissa tiimi esittelee toteutetun tuoteinkrementin. Tapahtumaan osallistuu asiakkaan edustajia, joihin tekeillä oleva ohjelmisto jollain tavalla vaikuttaa [Schwaber04, s. 137-138, s. 143]. Tämä tarkoittaa esimerkiksi yrityksen johtoa ja loppukäyttäjiä. Esitystä seuraamassa olleilta ihmisiltä kysytään heidän vaikutelmaansa tuotteesta, ja he voivat pyytää korjauksia tai uutta toiminnallisuutta lisättäväksi tuotteen työlistaan. Näin mahdolliset puutteet ja ongelmat pyritään saamaan kiinni.

Käyttöliittymän arvioinnin kannalta menetelmä on riskialtis. Käyttäjien mielipiteiden kysyminen ei ole luotettava menettely ongelmien löytämiseen [Nielsen01]. Käyttäjien ehdottamat toiminnot voivat olla huonoja ratkaisuja varsinaiseen ongelmaan, sillä he eivät tyypillisesti tiedä, minkälaiset ratkaisut olisivat mahdollisia [Cooper95 s. 549]. On olemassa lukuisia esimerkkejä ohjelmistoista, joiden

ongelmallisen käyttöliittymän syy on käyttäjien pyyntöjen kritiikitön toteuttaminen [Cooper95 s. 549, Norman05]. Katselmointitilaisuudessa esitystä seuraavien asiakkaan edustajien joukossa ei tyypillisesti ole käyttöliittymäsuunnittelun ja -arvioinnin asiantuntijoita. Projektin liiketoiminnalliseen kannattavuuteen keskittyvällä tuotteen omistajallakaan ei yleensä ole tätä osaamista [Singh08]. Valmius todellisten käytettävyysongelmien paikantamiseen sprintin katselmoinnissa on huono.

Kun käyttöliittymän arvioijalta puuttuu tarvittava asiantuntemus, eivät arvioinnin tulokset ole luotettavia [Desurvire92]. Todellisten ongelmien sijaan saadaan usein virheellisiä vaatimuksia korjauksille, joita ei pitäisi koskaan tehdä (*false alarms*). Asiakas saattaa esimerkiksi pyytää, että lentohakujärjestelmän hakutuloslistassa näytettävät välilaskujen vaihtoajat siirretään lisätietonäytölle, koska ”hakutulos-listassa on liikaa tietoa”. Jos pyyntö toteutettaisiin, joutuisi loppukäyttäjä katsomaan lisätietonäytön jokaisen kiinnostavan hakutuloksen kohdalla, jolloin hakutuloslistan ja lisätietonäytön välillä navigointi aiheuttaisi turhia toimenpiteitä ja hankaloittaisi hakutulosten vertailua. Asiakkaan pyynnöstä seuraisi tehokkuusongelma käyttöliittymään. Virheellisten korjausten toteuttaminen sekä huonontaa käyttö-liittymäratkaisua että kuluttaa tarpeettomasti toteutustyön resursseja.

Katselmoinnissa on kuitenkin mahdollista löytää hyödyllisyyspuutteita. Esitystä seuraamaan tarvitaan sellaisia loppukäyttäjiä, joiden työtehtäviä järjestelmällä simuloidaan. Käyttäjien nähdessä omien työtehtäviensä suorittamista järjestelmällä heidän on mahdollista paikantaa puutteita suhteessa työtehtäviinsä. Katselmointi-asetelma vastaisi hieman luvussa 3.3.4 esiteltyä hyödyllisyysläpikäyntiä. Tällöin vähintään yhdellä tiimin jäsenellä pitää olla osaamista hyvästä demotekniikasta eli järjestelmän simuloinnista oikeilla käyttötilanteilla ja realistisella esimerkkidatalla [Laakso05, s. 238]. Simulointia varten demon esittäjän täytyy tuntea loppukäyttäjien työtehtävät riittävän hyvin ja tuoteinkrementissä täytyy olla toteutettuna sellainen joukko toiminnallisuutta, että työtehtävien suorittaminen järjestelmällä on mahdollista. Tyypillisessä sprintin katselmoinnissa mainitut edellytykset eivät täyty ja riski hyödyttömille tai virheellisille tuloksille kasvaa. Jos simuloinnin sijaan tiimi esittelee irrallisia toimintoja, loppukäyttäjien on vaikea kuvitella, miten järjestelmällä

tullaan suorittamaan työtehtäviä. Tällöin he kommentoivat usein merkityksettömiä yksityiskohtia, kuten värejä tai ikoneita [Beyer98, s. 368].

Vaikka katselmointiin osallistuvilla henkilöillä olisi hyvä osaaminen käyttöliittymien arviointia varten, on tuoteinkrementin valmiin koodin muuttaminen erittäin kustannustehoton menettely verrattuna paperiprototyyppien muuttamiseen. Paperiprototyyppien laatiminen on erittäin nopeaa, ja Nielsenin esittämän arvion mukaan muutoksen tekeminen paperiprototyyppiin voi olla jopa sata kertaa edullisempaa kuin valmiin koodin muuttaminen [Nielsen03].

5 Käyttöliittymäriskien minimointi GDD-suunnittelulla

Ohjelmistoyritys Reaktorissa on jo vuosien ajan käytetty GDD-menetelmää käyttöliittymäsuunnitteluun ja Scrum-prosessimallia toteutukseen. Ammattikorkeakoulujen toiminnansuunnittelujärjestelmän (Toisu) kehittäminen oli ensimmäinen projekti, jossa GDD ja Scrum pyrittiin järjestelmällisesti yhdistämään. Reaktor oli ennen yhdistämistä jo toteuttanut Toisu-järjestelmästä yhden version, jonka vuosisuunnitteluosio osoittautui käyttöönoton jälkeen ongelmalliseksi. Vuosisuunnittelun työkalu toteutettiin alusta asti uudestaan erillisessä projektissa, jossa päästiin hyvään lopputulokseen GDD-käyttöliittymäsuunnittelua ja Scrumia käyttäen. Projektin avulla tarkastellaan, miten Scrum-prosessimallin käyttöliittymäriskejä minimoitiin simulointipohjaisella GDD-menetelmällä.

Tätä työtä varten on haastateltu Reaktorin käyttöliittymäsuunnittelijoita Karri-Pekka Laaksoa ja Vesa-Matti Mäkistä, jotka olivat mukana vuosisuunnittelun uuden version kehitysprojektissa alusta loppuun. Haastattelussa käytiin läpi Toisu-projektin kulku kronologisessa järjestyksessä, ja kaikista eteen tulleista tutkielman kannalta kiinnostavista kohdista pyydettiin konkreettisia esimerkkejä. Haastattelun jälkeen pyydettiin kopiot projektin tuottamista käyttöliittymäkuvauksista sekä tuotteen ja sprinttien työlistoista.

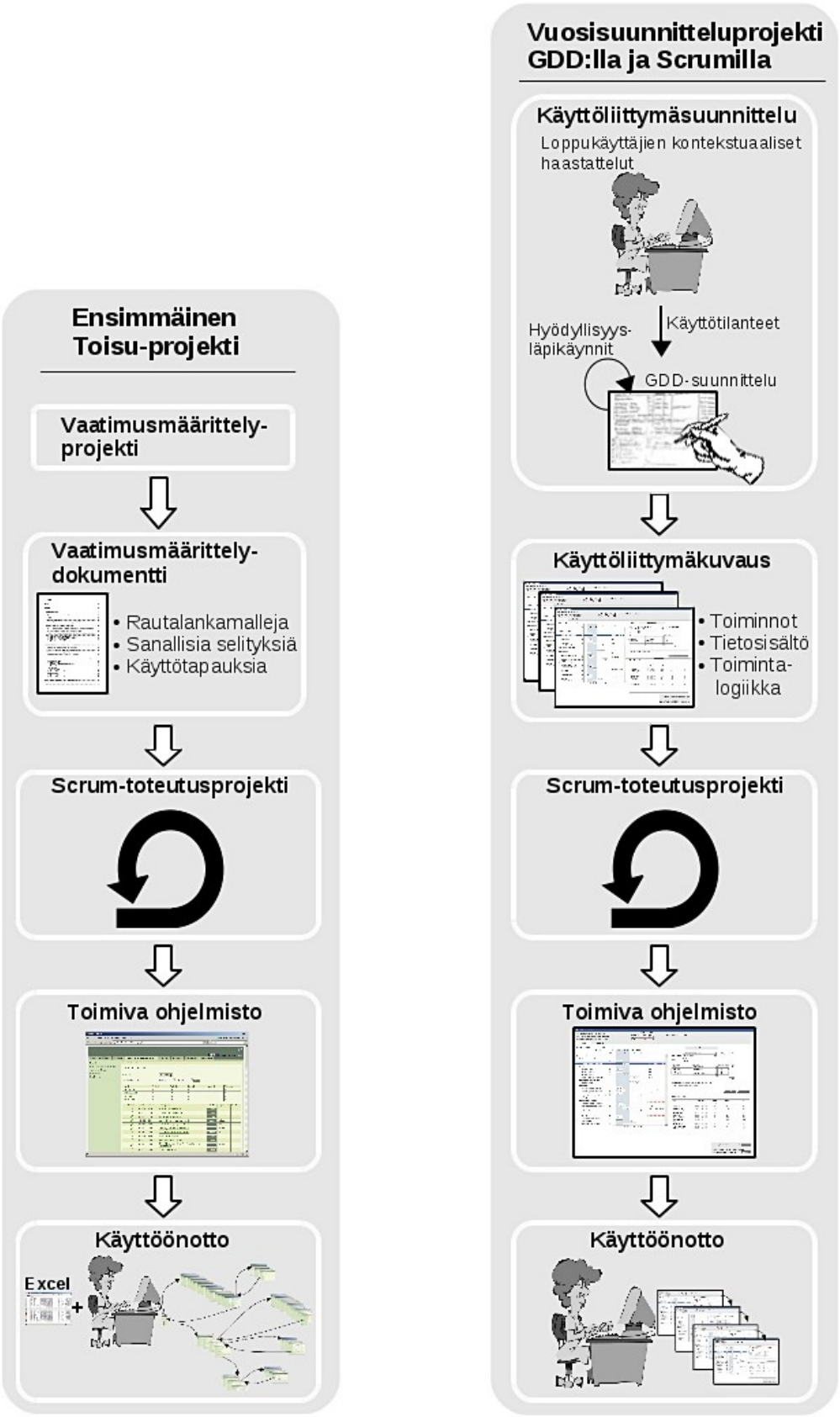
Luvussa 5.1 käsitellään uuden vuosisuunnittelun version kehitysprojekti. Ensin tarkastellaan vanhan Toisu-järjestelmän ongelmia ja vuosisuunnittelun uuden version GDD-käyttöliittymäsuunnittelua. Lopuksi käsitellään käyttöliittymäratkaisun toteuttaminen Scrum-menetelmällä: miten toteutustyö tehtiin GDD:n käyttöliittymäkuvauksen perusteella, millaisiin ongelmiin törmättiin ja miten ongelmat ratkaistiin. Luvussa 5.2 tarkastellaan ensin, miten Scrum-menetelmän käyttöliittymäriskit on minimoitu vuosisuunnittelun uuden version kehittämisessä. Tämän jälkeen käsitellään projektin kuluessa eteen tulleet Scrumista riippumattomat käyttöliittymäriskit. Luvussa 5.3 käsitellään GDD:n tuomat muutokset Scrumin rooleihin ja käytäntöihin.

5.1 Vuosisuunnitteluosion kehitysprojekti

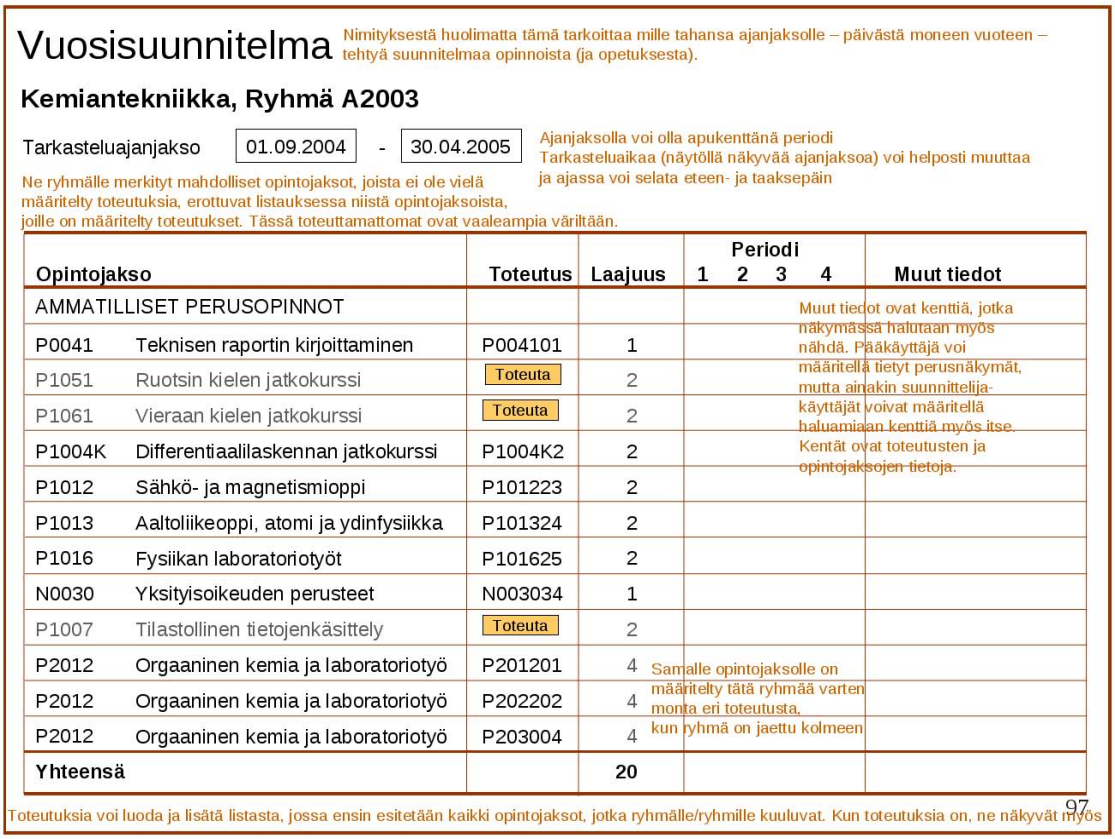
Reaktor sai ensimmäisen Toisu-version toteutuksen syötteeksi erään ohjelmisto-yrityksen laatiman vaatimusmäärittelydokumentin, jossa järjestelmän toiminnallisuus ja tietosisältö oli kuvattu sanallisilla selityksillä, rautalankamalleilla ja käyttö-tapauksilla (kuvan 5.1 vasemman palkin yläosa). Reaktor teki toteutuksen Scrum-menetelmällä, ja ohjelmiston käyttöliittymä syntyi pääosin rautalankamallien perusteella. Rautalankamalleilla tarkoitetaan tässä yhteydessä alustavia käyttö-liittymän näyttökuvia, joissa osa tiedoista ja toiminnoista on piirretty näkyviin ja osa jätetty sanallisten selitysten varaan. Kuvassa 5.2 on esimerkki rautalankamallin näyttökuvasta. Ensimmäisen Toisu-version toteutuksessa ei ollut mukana käyttö-liittymäsuunnittelijoita.

Reaktor toimitti asiakkaalle vaatimusmäärittelyn mukaisen ohjelmiston ja järjestelmä otettiin käyttöön. Käyttöönoton jälkeen asiakas reklamoi vuosisuunnitteluosiosta: ”Tämä ei ole suunnittelujärjestelmä, koska sillä ei voi suunnitella.” Reaktorin käyttöliittymäsuunnittelijat ryhtyivät selvittämään, minkä takia vuosisuunnitteluosio ei ollut käyttökelpoinen. Jo ennen vaatimusmäärittelydokumenttia asiakas oli teettänyt työnkulkuselvityksiä, joiden perusteella Reaktorin käyttöliittymä-suunnittelijat pystyivät simuloimaan työnkulkuja Toisu-järjestelmällä. Vuosi-suunnitteluosiosta paljastui tehokkuusongelmia, ja käyttöliittymäsuunnittelijat suoraviivaistivat ilmi tulleita monivaiheisia työnkulkuja. Muutosten teosta aiheutui kuitenkin niin isoja ongelmia taustajärjestelmän teknisen toteutuksen kanssa, että Reaktor päätti lopulta luopua vanhan järjestelmän korjaamisesta.

Asiakkaan kanssa sovittiin uudesta määrittelyprojektista, jossa laadittaisiin toimiva käyttöliittymä vuosisuunnittelun käyttötilanteisiin. Määrittelyprojekti tehtiin loppukäyttäjien eli ammattikorkeakoulujen koulutuspäälliköiden kontekstuaalisilla haastatteluilla ja GDD-käyttöliittymäsuunnittelulla (kuvan 5.1 oikean palkin ylin laatikko). Koulutuspäällikön tehtävänä on ajoittaa ja resursoida ammattikorkea-koulun opintosuunnitelman mukaiset opintojaksojen toteutukset. Hän huolehtii myös opiskelijoiden ja opettajien tasaisesta työkuormasta ja budjetissa pysymisestä.



Kuva 5.1: Ensimmäisen Toisu-projektin ja vuosisuunnittelun uuden version kehittämisen vaiheet.



Vuosisuunnitelma

Nimityksestä huolimatta tämä tarkoittaa mille tahansa ajanjaksolle – päivästä moneen vuoteen – tehtyä suunnitelmaa opinnoista (ja opetuksesta).

Kemiantekniikka, Ryhmä A2003

Tarkasteluajanjakso 01.09.2004 - 30.04.2005

Ajanjaksolla voi olla apukenttänä periodi
Tarkasteluaikaa (näytöllä näkyvää ajanjaksoa) voi helposti muuttaa
ja ajassa voi selata eteen- ja taaksepäin

Ne ryhmälle merkityt mahdolliset opintojaksot, joista ei ole vielä määritelty toteutuksia, erottuvat listauksessa niistä opintojaksoista, joille on määritelty toteutukset. Tässä toteuttamattomat ovat vaaleampia väriltään.

Opintojakso		Toteutus	Laajuus	Periodi 1 2 3 4	Muut tiedot
AMMATILLISET PERUSOPINNOT					
P0041	Teknisen raportin kirjoittaminen	P004101	1		
P1051	Ruotsin kielen jatkokurssi	Toteuta	2		
P1061	Vieraan kielen jatkokurssi	Toteuta	2		
P1004K	Differentiaalilaskennan jatkokurssi	P1004K2	2		
P1012	Sähkö- ja magnetismioppi	P101223	2		
P1013	Aaltoliikeoppi, atomi ja ydinfysiikka	P101324	2		
P1016	Fysiikan laboratoriotyöt	P101625	2		
N0030	Yksityisoikeuden perusteet	N003034	1		
P1007	Tilastollinen tietojenkäsittely	Toteuta	2		
P2012	Orgaaninen kemia ja laboratoriotyö	P201201	4		
P2012	Orgaaninen kemia ja laboratoriotyö	P202202	4		
P2012	Orgaaninen kemia ja laboratoriotyö	P203004	4		
Yhteensä			**20**		

Muut tiedot ovat kenttiä, jotka näkymässä halutaan myös nähdä. Pääkäyttäjä voi määritellä tietyt perusnäkymät, mutta ainakin suunnittelija-käyttäjät voivat määritellä haluamiaan kenttiä myös itse. Kentät ovat toteutusten ja opintojaksojen tietoja.

Samalle opintojaksolle on määritelty tätä ryhmää varten monta eri toteutusta, kun ryhmä on jaettu kolmeen

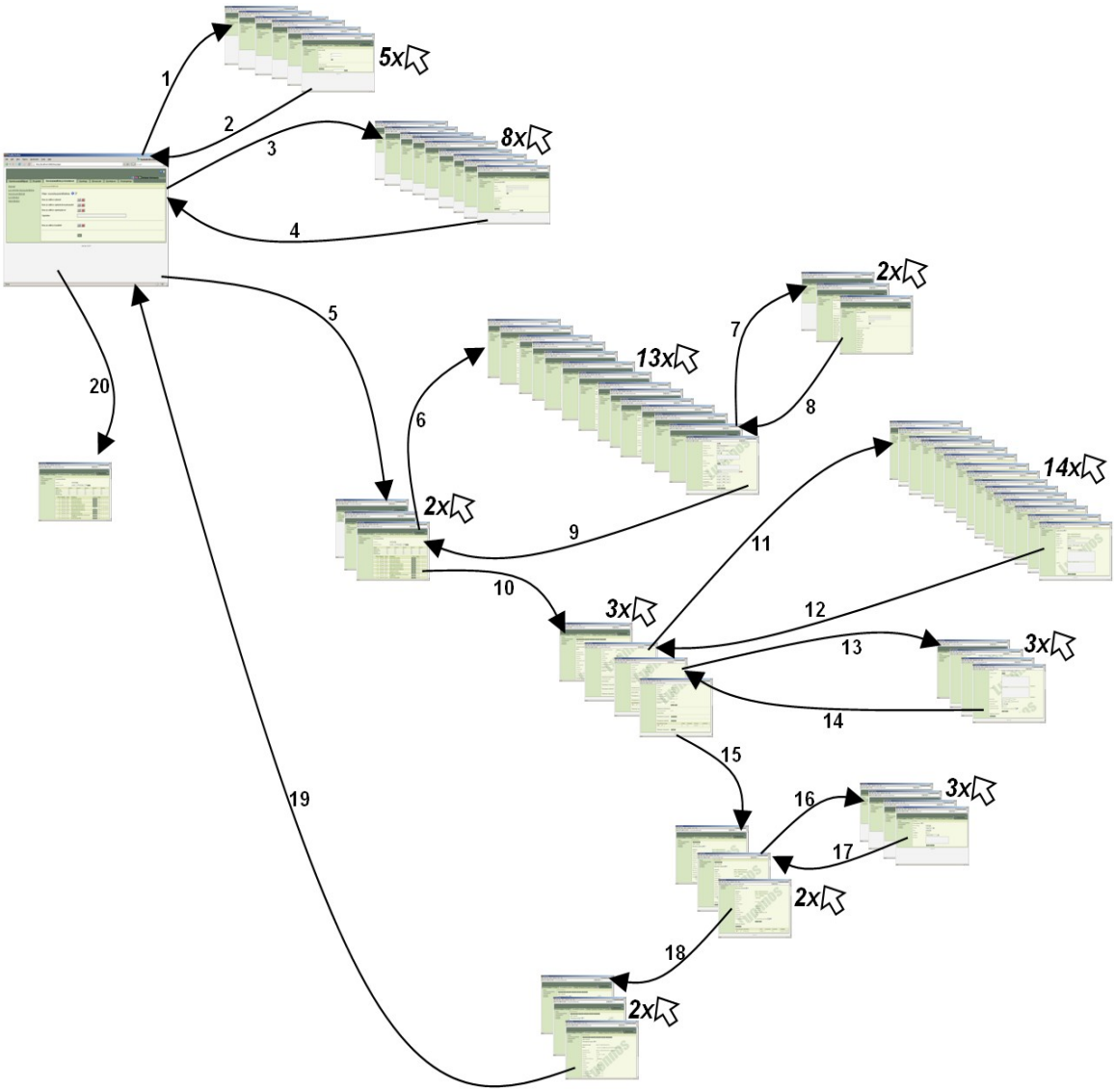
97

Toteutuksia voi luoda ja lisätä listasta, jossa ensin esitetään kaikki opintojaksot, jotka ryhmälle/ryhmille kuuluvat. Kun toteutuksia on, ne näkyvät myös

Kuva 5.2: Yksi kuva toteutuksen syötteenä olleesta rautalankamallista [Stadia04].

Kontekstuaalisissa haastatteluissa käyttöliittymäsuunnittelijat näkivät, kuinka ammattikorkeakoulujen koulutuspäälliköt tekivät työtään. Ongelmien vakavuus selvisi lopullisesti vasta haastatteluissa: Ennalta tiedossa olleiden puutteiden lisäksi tuli ilmi, että järjestelmästä puuttui keskeistä suunnittelussa tarvittavaa tietosisältöä, kuten käytettävissä oleva budjetti ja kustannukset. Koulutuspäälliköt tekivät suunnittelutyön Excelin avulla, ja Toisu-järjestelmää käytettiin suunnittelutyön tulosten mekaaniseen kirjaamiseen. Mekaaninen kirjausvaihe tarvittiin, jotta opettajien työkuorma saatiin tallennettua palkanmaksua varten ja opiskelijoiden ilmoittautuminen ja arvostelu saatiin hoidettua.

Kuvassa 5.3 näytetään yhden kurssitoteutuksen käsittelyä ongelmallisella vuosi-suunnitteluosiolla. Kuvan tilanteessa koulutuspäällikkö on jo tehnyt varsinaisen suunnittelutyön Excelissä ja kuvatussa sekvenssissä hän kiinnittää yhdelle kurssitoteutukselle vastuuhenkilön, luennoijan ja 20 tuntia luentoja. Toimenpide-sekvenssissä on yhteensä 77 toimenpidettä ja 20 ikkunasiirtymää. Koko lukuvuoden



Kuva 5.3: Yhden kurssitoteutuksen käsittelyä ongelmallisella vuosisuunnittelulla (näyttökuvat poimittu lähteestä [Mäkinen07]).

kurssitoteutusten syöttäminen järjestelmään vaatii saman sekvenssin toistamisen yli 100 kertaa. Vuosisuunnittelun tehokkuusongelma oli varsin paha.

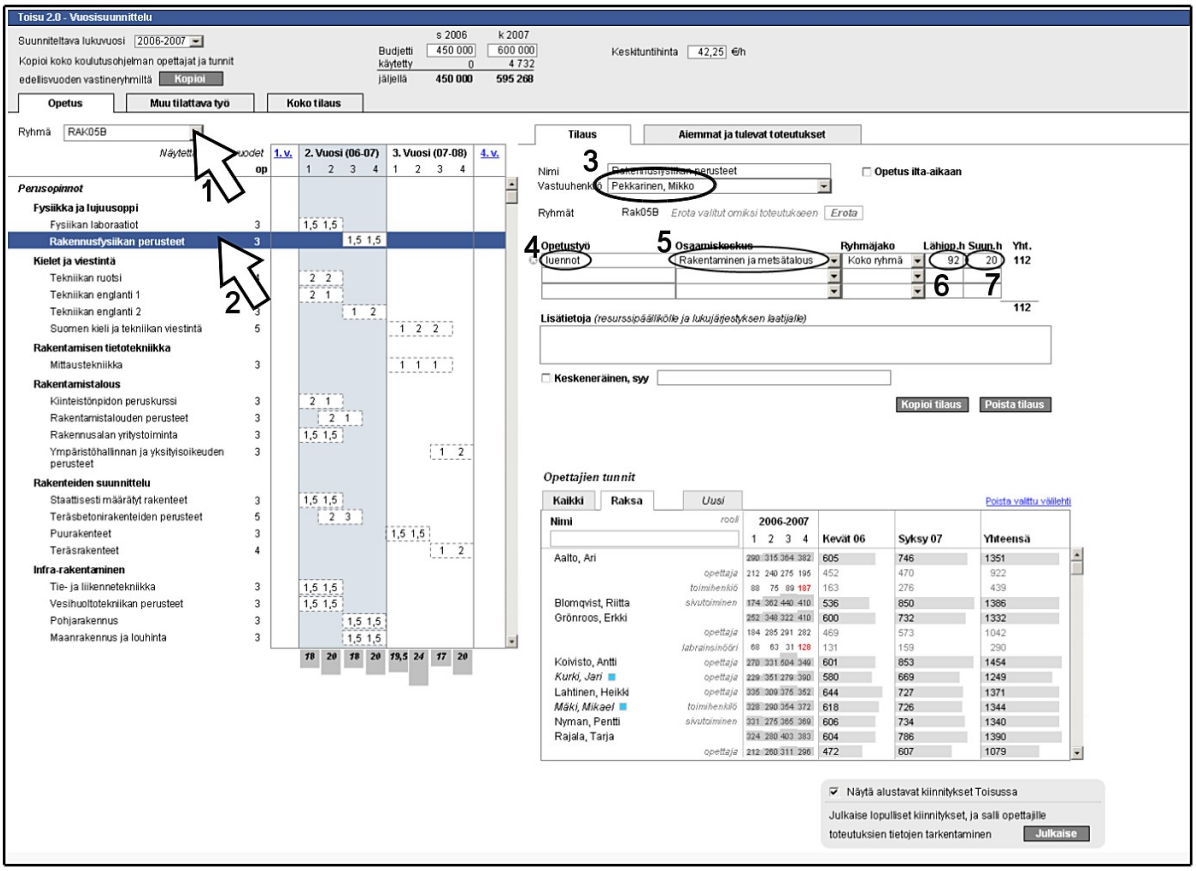
Käyttäjähaastattelujen jälkeen käyttöliittymää alettiin suunnitella GDD-menetelmällä alusta lähtien uusiksi (kuvan 5.1 oikean palkin ylin laatikko). Käyttöliittymäsuunnittelijat simuloivat haastatteluissa selvitettyjä käyttötilanteita Reaktorin toimistolla ja piirsivät puhtaaksi paperiprototyypin käyttöliittymän ensimmäisestä versiosta. Sen jälkeen he menivät koulutuspäällikön luokse ja paperiprototyypillä tehtiin hyödyllisyysläpikäynti hänen kanssaan. Läpikäynnissä ilmi tulleet puutteet korjattiin käyttöliittymään, uudet esille tulleet käyttötilanteet otettiin mukaan GDD-

suunnitteluun ja suunnittelun jälkeen paperiprototyypistä piirrettiin puhtaaksi uusi versio seuraavaa hyödyllisyysläpikäyntiä varten. Läpikäynneissä selvisi esimerkiksi se, että toisin kuin oli luultu, muutokset jo resursoituun opetukseen lukuvuoden aikana eivät olleet lainkaan harvinaisia. Opintojaksolle saattoi kesken lukuvuoden tulla uutena asiana esimerkiksi 40 tuntia kieliharjoituksia. Käyttöliittymän ensimmäiset versiot eivät vielä täysin tukeneet tällaisten muutosten tekoa ja puutteet korjattiin käyttöliittymään läpikäynnin jälkeen. Käyttöliittymäsuunnittelu jatkui edellä kuvatulla tavalla ja paperiprototyypin viimeinen versio valmistui neljännen hyödyllisyysläpikäynnin jälkeen. Paperiprototyypistä laadittiin tietokoneella puhtaaksi piirretty käyttöliittymäkuvaus, jossa käyttöliittymäratkaisu oli sovitettu 1400 pikseliä leveään näyttöresoluutioon ja lopullinen ulkoasu oli kiinnitetty.

Kun kuvan 5.3 sekvenssissä on 77 toimenpidettä ja 20 ikkunasiirtymää, uudella käyttöliittymäratkaisulla vastaava sekvenssi vaatii seitsemän toimenpidettä ilman ikkunasiirtymiä (kuva 5.4). Uudella käyttöliittymällä koulutuspäällikön on mahdollista tehdä suunnittelutyö, koska siinä tarvittava tietosisältö, kuten budjetti ja opettajien työkuorma, näytetään koulutuspäällikölle yhdellä kertaa.

Uusi käyttöliittymäratkaisu asetti haasteita toteutustekniikalle. Jo käyttöliittymä-suunnitteluvaiheessa ennen käyttöliittymäkuvauksen lopullista valmistumista nähtiin, ettei vanhan vuosisuunnittelun toteutuksessa käytetyllä selainpohjaisella tekniikalla pystyttäisi toteuttamaan määriteltyjä interaktioita riittävän luotettavasti. Esimerkiksi visualisointipalkkien venyttämistä hiirellä raahaamalla ei olisi voitu toteuttaa tarpeeksi laadukkaasti vuonna 2006 käytössä olleilla tekniikoilla. Selainpohjaisen sovelluksen sijaan vuosisuunnittelulle päädyttiin tekemään internetin yli käynnistettävä Java-sovellus (Java Web Start), joka mahdollisti käyttöliittymän interaktioiden toteuttamisen ilman kompromisseja. Näin loppukäyttäjille päätyisi paras mahdollinen ratkaisu ilman etukäteen valitun liian jäykän toteutustekniikan rajoitteita.

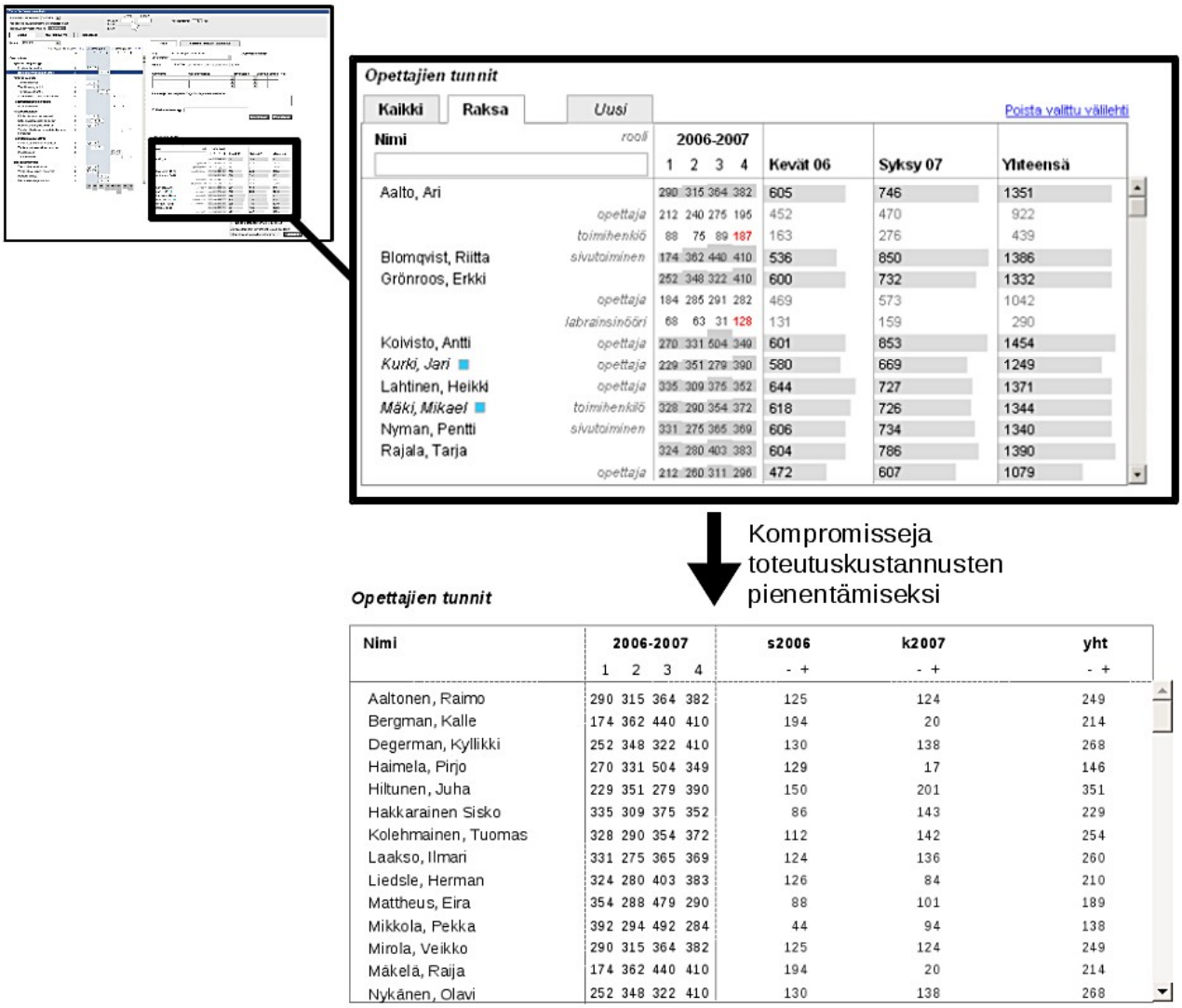
Toteutustiimi laati käyttöliittymäkuvauksen perusteella kustannusarvion, joka oli asiakkaalle liian kallis. Järjestelmästä piti karsia ominaisuuksia, jotta toteutustyö ei ylittäisi resursseja. Käyttöliittymästä päätettiin toteuttaa vain oleellinen ja jättää pois



Kuva 5.4: Yhden kurssitoteutuksen käsittelyä vuosisuunnittelun uudella käyttöliittymällä (näyttökuva lähteestä [Reaktor06]).

tuki epätyypillisemmille käyttötilanteille, sekä korvata osa hankalammin toteutettavista ratkaisuista yksinkertaisemmilla. Tässä tilanteessa oli riski, että poistetut ominaisuudet estäisivät myös joidenkin tuettavien käyttötilanteiden suorittamisen tai että karsiminen heikentäisi järjestelmän käytettävyyttä liikaa.

Esimerkki tehdystä kompromissista näytetään kuvassa 5.5, jossa opettajien työtuntilistasta on poistettu mm. taulukon sarakkeissa *Kevät 06*, *Syksy 07* ja *Yhteensä* olevat työkuormaa visualisoivat palkit. Visualisointipalkkien avulla käyttäjä pystyy hahmottamaan kokonaiskuvan työkuorman jakautumisesta opettajien kesken yhdellä silmäyksellä, mutta ilman visualisointipalkkeja hän joutuu lukemaan kaikki numerot läpi ja pitämään niitä mielessään vertailemisen aikana muodostaakseen kokonaiskuvan. Kun visualisointipalkkeja ei ole, käyttäjältä vaadittava mentaalityö lisääntyy ja siten käytettävyys kärsii, mutta käyttäjä pystyy silti tekemään työn.



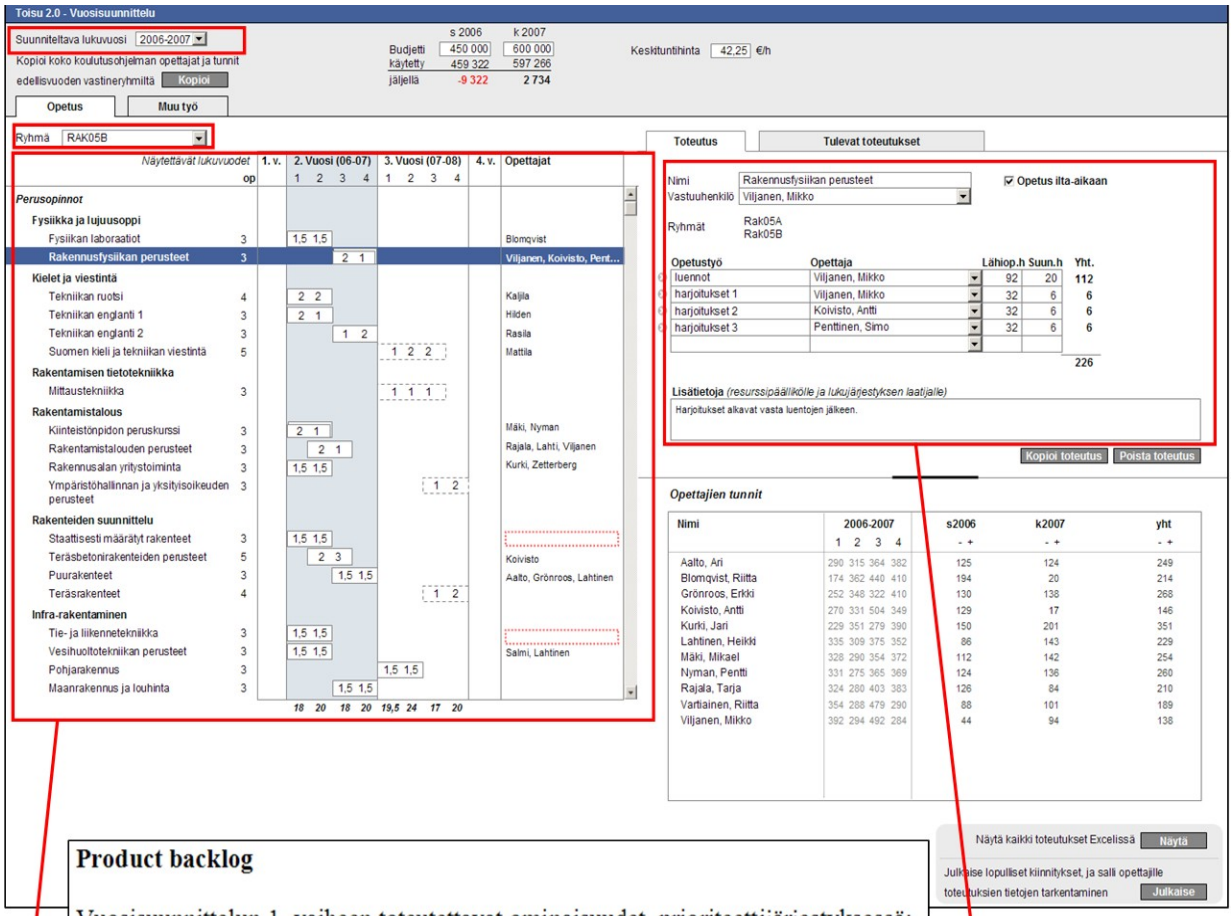
Kuva 5.5: Esimerkki kompromissiratkaisusta kustannusten pienentämiseksi (näyttökuvat lähteestä [Reaktor06]).

Käyttöliittymäsuunnittelijat suorittivat ominaisuuksien karsimisen varmistaen simuloimalla, että kaikki tuettavat käyttötilanteet pystyy suorittamaan myös karsitulla järjestelmällä alusta loppuun asti. Käyttöliittymäkuvauksesta tehtiin kompromissiversio.

Vertaamalla käyttötilanteiden suorittamista optimi- ja kompromissikäyttöliittymällä asiakas voi nähdä, miten kustannusten perusteella karsitut ominaisuudet vaikuttavat loppukäyttäjän työtehtävien suorittamiseen. Samalla asiakkaalle jää tieto siitä, millaisia työtä tehostavia ominaisuuksia resursseja lisäämällä voidaan saada. Esimerkiksi projektin saadessa lisärahoitusta optimikäyttöliittymäkuvaus auttaa

arvioimaan, mihin kompromissiratkaisusta pois jätettyihin ominaisuuksiin rahat kannattaa käyttää.

Ennen toteutussprinttejä käyttöliittymäsuunnittelijat ja toteutustiimi laativat yhdessä tuotteen työlistan poimimalla kompromissikäyttöliittymään määritellyt ominaisuudet riveiksi työlistaan. Kuvassa 5.6 on esimerkki kahdesta käyttöliittymäratkaisun perusteella tuotteen työlistaan poimitusta ominaisuudesta. Kuvasta on alleviivattu 2 työlistan riviä ja niitä vastaavat käyttöliittymän osat on korostettu.



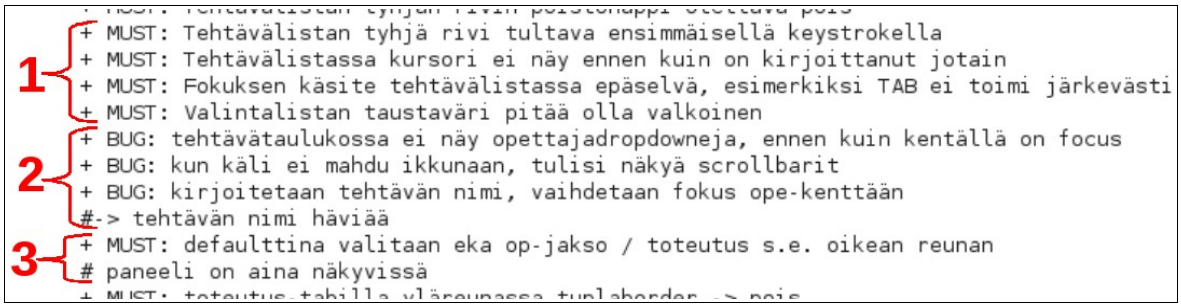
Kuva 5.6: Tuotteen työlista käyttöliittymäkuvauksen perusteella (kuvat lähteestä [Reaktor06]).

Käyttöliittymäsuunnittelijat priorisoivat työlistan: kenttätutkimusten ja GDD-käyttöliittymäsuunnittelun jälkeen heillä oli hyvä käsitys tarvittavien toimintojen tärkeydestä suhteessa tuettaviin työtehtäviin. Siten käyttöliittymäsuunnittelijat ottivat käytännössä Scrum-menetelmän tuotteen omistajan roolin työlistan luomisessa ja priorisoinnissa.

Alkuperäisessä Scrum-menetelmässä asiakasta edustava tuotteen omistaja on vastuussa työlistan ominaisuuksista. Toisu-projektissa tämä ei olisi ollut järkevä menettely, sillä tarvittava toiminnallisuus, tietosisältö ja toimintalogiikka oli jo selvitetty kenttätutkimusten ja GDD-käyttöliittymäsuunnittelun avulla. Ongelmana oli enää määrittelyn mukaisen ohjelmiston toteuttaminen. Tämän ongelman ratkaisu oli toteutustiimin ja suunnitellun järjestelmän toimintalogiikan tuntevien käyttöliittymäsuunnittelijoiden vastuulla. Asiakkaan rooli oli hyväksyä tehty työ.

Toteutussprintit aloitettiin ja käyttöliittymäsuunnittelijat demonstroivat käyttöliittymäkuvauksen toimintalogiikan toteutustiimille sprintin suunnittelukokouksessa. Tiimi teki toteutustyötä itsenäisesti sprinttien ajan ja toteuttajat kysyivät ajoittain heille epäselvien kohtien toimintalogiikasta käyttöliittymäsuunnittelijoilta. Toteutettu toimintalogiikka ei kuitenkaan täysin vastannut määriteltyä logiikkaa. Ohjelmoinnissa sattui väärinymmärryksiä logiikasta ja lisäksi joitain interaktioiden yksityiskohtia oli toteutettu puutteellisesti.

Esimerkkejä puutteista näytetään kuvassa 5.7, jossa neljännen sprintin työlistasta on korostettu eri tyyppisiä tuoteinkrementissä ilmenneitä ongelmia. Kuvan kohdassa 1 on esimerkkejä puutteista interaktioiden yksityiskohdissa, kuten tekstikursorin puuttuminen. Kohdan 2 puutteet vaikuttavat olevan toteutukseen jääneitä virheitä, kuten jo syötetyn tiedon häviäminen kentän fokusta vaihdettaessa. Kohdassa 3 on esimerkki väärinymmärryksestä toimintalogiikassa: Toteutetussa käyttöliittymässä kuvan 5.6 vasemman puolen opintojaksolistasta puuttui oletusvalinta ja oikean puolen paneeli ilmestyi näkyviin vasta käyttäjän valinnan jälkeen. Overview beside Detail -suunnittelumallin (ObD) [Laakso04b, s. 1-2] tuntevalle käyttöliittymäsuunnittelijalle on itsestään selvää, että vasemman puolen listassa (overview) on aina



Kuva 5.7: Ote neljännen sprintin työlistasta, eri tyyppisiä puutteita tuotteessa korostettu (kuva lähteestä [Reaktor06]).

yksi opintojakso valittuna ja näytön oikean puolen paneeli (detail) on aina näkyvillä. Jos ObD-malli ei ole tuttu ohjelmoijalle, opintojaksolistan oletusvalinta ei ole itsestään selvä asia.

Puutteita päätyi tuotantovalmiiksi kuitattuun toiminnallisuuteen asti. Kolmen ensimmäisen sprintin aikana puutteita korjattiin samalla, kun toteutukseen otettiin uutta toiminnallisuutta. Kolmannen sprintin jälkeisessä tuoteinkrementissä oli kuitenkin niin paljon korjausta vaativia kohtia, että uutta toiminnallisuutta ei voitu ottaa neljänteen sprinttiin mukaan lainkaan. Tuoteinkrementti ei ollut käyttöliittymän interaktioiden osalta tuotantokelpoinen, ja neljäs sprintti päätettiin käyttää kokonaan puutteiden korjaamiseen.

Väärinymmärrykset, virheet ja puutteelliset interaktiot päätyivät tuoteinkrementtiin, sillä toteutustiimillä ja käyttöliittymäsuunnittelijoilla ei ollut käytössä systemaattista menettelyä toteutuksen toimintalogiikan tarkistamiseen. Kolmannen sprintin jälkeen erään toteuttajan aloitteesta projektissa otettiin käyttöön hyväksymismenettely. Jatkossa toteuttajat antaisivat kaikki toteutetut ominaisuudet käyttöliittymä-suunnittelijoiden hyväksyttäväksi. Vasta kun suunnittelijat olisivat käyttötilanteita simuloimalla tarkastaneet toteutuksen toimintalogiikan vastaavan määriteltyä, kuitattaisiin kyseinen ominaisuus tehdyksi. Hyväksymismenettelyn käyttöönoton jälkeen ohjelmoijat oppivat nopeasti, millaista laatua käyttöliittymätoteutukselta vaaditaan. Toteutustyön laatu parani huomattavasti: kun neljännen sprintin työlistassa oli 22 toteutuksesta korjattavaa kohtaa, niitä oli viidennen sprintin työlistassa enää 5 ja kuudennen 0.

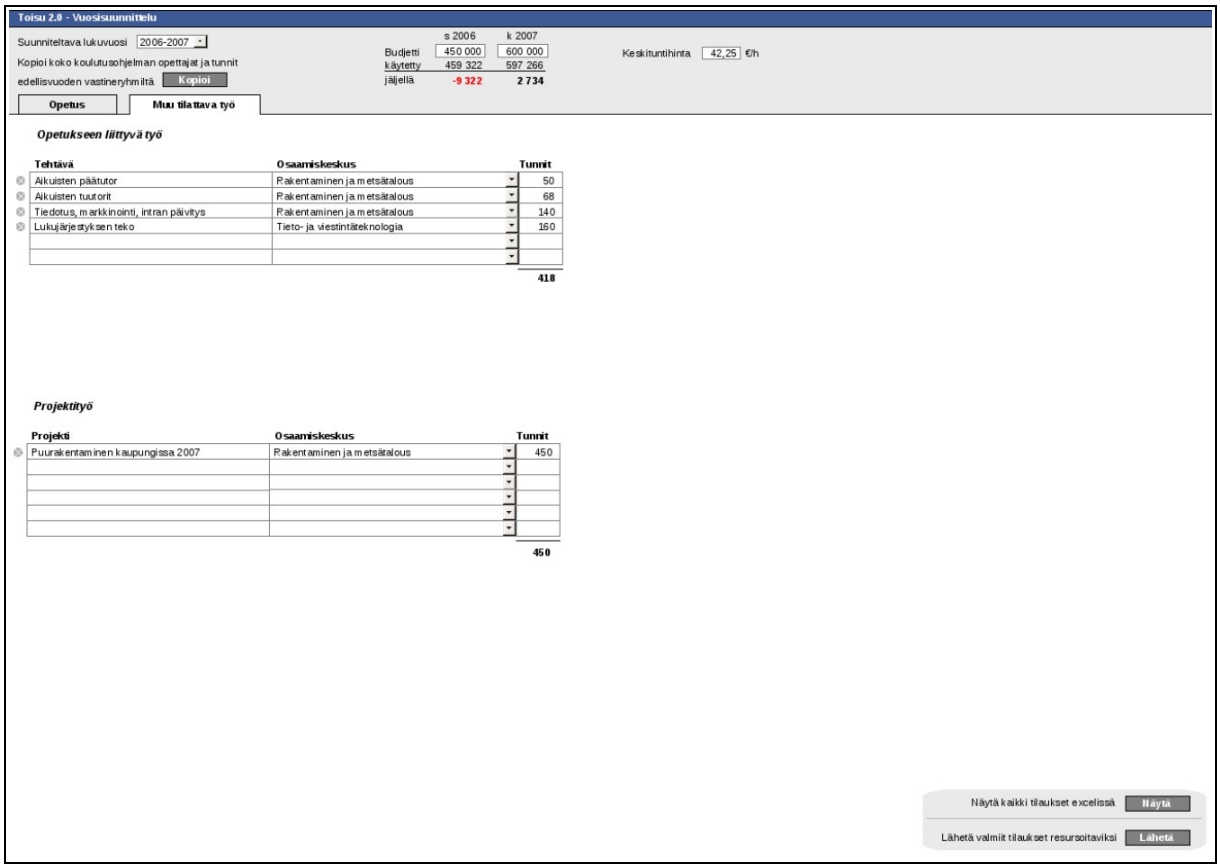
Hyväksymismenettely selkeytti tehtäväjakoa: Ohjelmoijat oppivat, että törmätessään heille epäselvään kohtaan toimintalogiikassa heidän ei tarvitse eikä pidä ratkaista sitä itse. Sen sijaan he voivat vaatia käyttöliittymäsuunnittelijoilta vastaukset käyttöliittymän toimintalogiikkaan liittyviin kysymyksiinsä. Jatkossa toteuttajat kysyivät epäselvistä kohdista aktiivisemmin käyttöliittymäsuunnittelijoilta, mikä vähensi toteutukseen päätyneiden puutteiden määrää. Tällä tavalla hyväksymismenettely paransi toteutustiimin ja käyttöliittymäsuunnittelijoiden välistä kommunikaatiota.

Hyväksymismenettely määritteli käyttöliittymäsuunnittelijoiden työnkuvan projektissa täsmällisemmin. Epämääräisemmästä roolista siirryttiin tilanteeseen, jossa toteuttajien ja käyttöliittymäsuunnittelijoiden työllä oli tiukka linkitys toisiinsa: toteuttajan saatua toiminnallisuuden valmiiksi käyttöliittymäsuunnittelijan tehtävänä oli tarkistaa toimintalogiikka mahdollisimman pian, jotta toteuttaja pääsisi nopeasti siirtymään seuraavaan ominaisuuteen.

Hyväksymismenettelyn ansiosta projektissa voitiin jatkossa luottaa siihen, että kaikki hyväksytyksi kuitatut ominaisuudet olivat tuotantovalmiita myös käyttöliittymän interaktioiden osalta. Esimerkiksi järjestelmää asiakkaalle demonstroidessa tiedettiin, että kaikki hyväksytyt ominaisuudet toimivat niin kuin pitääkin ja niiden voitiin luvata olevan sataprosenttisesti valmiita.

Hyväksymismenettelyn käyttöönoton jälkeen projekti eteni ongelmitta useamman sprintin ajan, kunnes ohjelmoijat pyysivät käyttöliittymäsuunnittelijoita demonstroimaan seuraavassa iteraatiossa toteutettavan näkymän toimintalogiikan. Selvisi, että käyttöliittymäkuvaukseen oli jäänyt keskeneräinen puhtaaksi piirretty kuva. Käyttöliittymäsuunnittelijat olivat laatineet kuvan alunperin asiakasdemoa varten, mutta he eivät olleet merkanneet kyseisen käyttöliittymän osan olevan keskeneräinen ja kuva päätyi lopulliseen käyttöliittymäkuvaukseen. Suunnittelun viemiseksi loppuun oli tehtävä vielä yksi käyttäjähaastattelu. Suunnitteluvaiheessa tapahtunut virhe aiheutti riskin puutteellisen ratkaisun päätymisestä toteutukseen. Riskin toteutuessa seurauksena olisi ratkaisun korjaamisesta aiheutuvaa turhaa ohjelmointityötä.

Kuvassa 5.8 on näyttökuva lähes toteutukseen päätyneestä keskeneräisestä käyttöliittymäratkaisusta. Ohjelmoija pystyisi laatimaan kuvan perusteella toteutuksen kiinnittämällä interaktioiden yksityiskohdat parhaan näkemyksensä mukaan. Jos tiimi olisi toteuttanut näkymän, keskeneräisyys olisi tullut ilmi vasta sitten, kun toteutus annettaisiin käyttöliittymäsuunnittelijalle hyväksyttäväksi. Näin ei päässyt käymään, sillä hyväksymismenettelyn käyttöönoton jälkeen ohjelmoijat vaativat käyttöliittymäsuunnittelijoilta vastaukset epäselviin kohtiin toiminta-logiikassa. Hyväksymismenettelyn myötä tulleen toteutustiimin ja käyttöliittymä-suunnittelijoiden välisen täsmällisen työnjaon ja aktiivisemman kommunikaation ansiosta ohjelmoija sai kiinni käyttöliittymäsuunnittelijoiden tekemän virheen. Scrum jousti tässä ongelmatilanteessa hyvin: Keskeneräinen toiminnallisuus siirrettiin prioriteetissa alemmaksi ja sprinttiin otettiin toteutettavaksi muita ominaisuuksia. Toteutustyö jatkui normaalisti ja tarvittava käyttäjähaastattelu sekä uusi käyttöliittymäratkaisu tehtiin kahden seuraavan sprintin aikana.



Kuva 5.8: Käyttöliittymäkuvaukseen erehdyksessä jäänyt keskeneräinen kuva [Reaktor06].

5.2 Käyttöliittymäriskien eliminointi vuosisuunnitteluprojektissa

Kuvan 4.1 riskikohta 1 (asiakkaan toivomat ominaisuudet tuotteen työlistaan vaatimuksiksi) minimoitiin laatimalla tuotteen työlista käyttöliittymäratkaisun perusteella. Sen sijaan, että asiakas olisi kirjannut toiveensa sellaisinaan vaatimuksiksi työlistaan, käyttöliittymäsuunnittelijat selvittivät koulutuspäälliköiden työtehtävät ja suunnittelivat käyttöliittymäratkaisun systemaattisesti simuloimalla selvitettyjä työtehtäviä. Käyttöliittymäratkaisun perusteella laadittu työlista sisälsi ominaisuudet, jotka oli suunniteltu tukemaan koulutuspäälliköiden työtä suoraviivaisesti ja testattu hyödyllisyysläpikäynneillä heidän kanssaan.

Kuvassa 4.1 esitetty riskikohta 2 (käyttöliittymän syntyminen tuotteen työlistan pohjalta ohjelmoinnin ohella) eliminoitiin suunnittelemalla vuosisuunnitteluosiosta kokonaan valmis käyttöliittymä ennen toteutusta. Koska toiminnot, tietosisältö ja toimintalogiikka olivat kiinnitetyt ennen toteutusvaihetta, toteutuksessa voitiin keskittyä määritellyn toimintalogiikan toteuttamiseen.

Kuvan 4.1 riskikohta 3 (sprintin katselmointi arviointimenettelynä) ratkaistiin arvioimalla käyttöliittymää neljällä hyödyllisyysläpikäynnillä, jolloin arviointi ei jäänyt sprintin katselmoinnin seuraajien esittämien mielipiteiden varaan. Epäluotettavat sprintin katselmoinnin tulokset korvattiin systemaattisella menettelyllä, jossa käyttöliittymäsuunnittelijat simuloivat koulutuspäälliköiden oikeiden töiden tekoa askel askeleelta heidän kanssaan.

Scrumin riskien eliminoinnin jälkeen projektiin jäi vielä Scrumista riippumattomia käyttöliittymäriskejä. Ensimmäinen vastaan tullut riski oli liian jäykkä toteutustekniikka. Toteutustekniikka vaihdettiin selainpohjaisesta Java Web Start -tekniikkaan, jolloin käyttöliittymäratkaisuihin ei tarvinnut tehdä käytettävyyttä heikentäviä kompromisseja. Toinen riskikohta oli toteutustyön arvioidut kustannukset, joiden pienentämiseksi käyttöliittymästä oli karsittava ominaisuuksia. Riskinä oli se, että poistetut ominaisuudet estäisivät joidenkin tuettavien käyttötilanteiden suorittamisen tai että karsiminen heikentäisi järjestelmän käytettävyyttä liikaa. Käyttöliittymäsuunnittelijat varmistivat käyttötilanteita simuloimalla, ettei

työtehtävien suorittaminen karsitulla järjestelmällä hankaloidu liikaa tai esty kokonaan.

Keskeisin Scrumista riippumaton riski tuli vastaan toteutussprinttien aikana. Käyttöliittymätoteutukseen päätyi puutteellisia interaktioita ja väärinymmärryksiä, jotka lopulliseen ohjelmistoon päätyessään olisivat heikentäneet sen käytettävyyttä oleellisesti. Siten kuvan 4.1 riskikohta 4 (lopullinen käyttäjälle päätyvä järjestelmä on ongelmallinen) oli vielä olemassa, vaikka käyttöliittymä suunniteltiin valmiiksi ennen toteutusvaihetta. Vuosisuunnitteluprojektissa riski eliminoitiin ottamalla käyttöön hyväksymismenettely, jossa käyttöliittymäsuunnittelijat tarkistivat toteutetun toimintalogiikan vastaavan määriteltyä. Hyväksymismenettely osoittautui tehokkaaksi keinoksi parantamaan käyttöliittymäsuunnittelijoiden ja toteutustiimin kommunikaatiota ja eliminoimaan puutteellisesti toimivan käyttöliittymätoteutuksen päätymisen valmiiseen tuoteinkrementtiin.

5.3 Scrumin roolien muuttuminen

GDD tuo Scrumiin uuden roolin, käyttöliittymäsuunnittelijan. Samalla tuotteen omistajan, tiimin ja Scrum-mestarin roolit muuttuvat. Tässä luvussa käsitellään käyttöliittymäsuunnittelijan tehtävät Scrumin käytännöissä ja muutokset Scrumin rooleihin.

Kuvassa 5.9 esitetään Scrum-prosessimalli, josta käyttöliittymäsuunnittelijan tehtävät on korostettu harmaalla taustalla. Alussa käyttöliittymäsuunnittelijoiden vastuulla on kenttätutkimusten suorittaminen, GDD-käyttöliittymäsuunnittelu ja käyttöliittymän testaus. Kun käyttöliittymäkuvaus on valmis, käyttöliittymäsuunnittelijat auttavat tiimiä ja tuotteen omistajaa tuotteen työlistan luomisessa ja priorisoinnissa.

Alkuperäisessä Scrumissa tuotteen työlistaan kirjataan toiminto- ja tietosisältövaatimukset. GDD-menetelmässä tarvittava toimintalogiikka ja tietosisältö selviää käyttöliittymäratkaisusta, joten työlista luodaan käyttöliittymäratkaisun perusteella. Työlistan sisältönä voi olla periaatteessa mitä tahansa, mutta listan tarkoitus on aina sama: auttaa toteuttamaan käyttöliittymäkuvaukseen määritelty toimintalogiikka.

Kuva 5.9: Käyttöliittymäsuunnittelijan tehtävät Scrumin käytännöissä (pohjaksi otettu kuva lähteestä [Sutherland07]).

Toisun vuosisuunnitteluosion toteuttamisessa työlistaan poimittiin käyttöliittymäratkaisusta näkyvät toiminnot ja tietosisältö. Tämä osoittautui toimivaksi ratkaisuksi. Ohjelmoijan kannalta käyttöliittymän pohjalta laadittu työlista saattaa näyttää hyvinkin samanlaiselta kuin alkuperäisen Scrumin työlista, mutta nyt työlistan ominaisuuksien toimintalogiikkaa ei kiinnitetä toteutussprinttien aikana. Sen sijaan toimintalogiikka on kiinnitetty käyttöliittymäratkaisussa.

GDD tuo uuden käsittelytavan myös projektin aikana tuleville muutos- ja toiminto-pyynnöille. Tuotteen omistaja ei enää lisää toiminto- tai muutospyyntöjä tuotteen työlistaan itse. Sen sijaan hän kääntyy käyttöliittymäsuunnittelijoiden puoleen, jotka selvittävät käyttäjähaastatteluilla tai -tarkkailuilla käyttötilanteet, joissa tuotteen omistaja on ajatellut toivomaansa ominaisuutta tarvittavan.

Jos pyydetylle ominaisuudelle ei löydy käyttötilanteita, asiakasta informoidaan tästä ja pyydetty ominaisuus jätetään sivuun. Jos käyttötilanteita löytyy, tuki tilanteille integroidaan käyttöliittymään simuloimalla GDD-menetelmällä. Koska käyttöliittymään laaditaan käyttäjän kannalta optimaalinen ratkaisu, käyttöliittymään

päätyvä ratkaisu voi olla eri kuin asiakkaan alunperin pyytämä. Tuotteen omistaja saattaa esimerkiksi pyytää, että julkisen liikenteen reittihakujärjestelmän hakutuloksiin lisättäisiin esteettömyystiedot, jotta liikuntarajoitteiset voisivat välttää korkealattiabussit ja esteelliset pysäkit. Käyttötilanteita simuloimalla saatetaan huomata, että pelkät esteettömyystiedot eivät ole käyttäjille kovin hyödyllisiä. Päästäkseen tavoittelemaansa paikkaan käyttäjän olisi myös pystyttävä hakemaan esteettömiä reittejä ja välttämään pysäkkien lisäksi muutkin reitille osuvat esteet, kuten portaat. Kun uusi käyttötilanne on integroitu käyttöliittymään, voidaan käyttötilanteen simuloinnin perusteella generoituneet ominaisuudet lisätä tuotteen työlistaan. Tällä tavalla saadaan varmistettua, että tuotteen työlistaan päätyvät ominaisuudet ovat oikeita ratkaisuja asiakkaan ongelmaan.

Edellä kuvatulla työnjaolla asiakkaan ei tarvitse ottaa vastuuta potentiaalisesti ongelmallisesta määrittelystä. Vastuu varsinaisen ongelman selvittämisestä ja ratkaisun laatimisesta jää käyttöliittymäsuunnittelijoille.

Tuotteen omistaja toki viime kädessä hyväksyy laaditun käyttöliittymäratkaisun ja hänellä voi olla edelleen iso rooli tuotteen työlistan priorisoinnissa, johon hän tuo mukaan yrityksen liiketoimintanäkökulmat. Hänen ei kuitenkaan kannata tehdä priorisointia yksin. Käyttöliittymäsuunnittelijat tietävät ominaisuuksien tärkeyden suhteessa tuettaviin käyttötilanteisiin. He osaavat esimerkiksi kertoa, mitkä ominaisuudet tarvitaan tärkeimpien työtehtävien suorittamiseen alusta loppuun asti [Laakso07]. Toteutustiimi tuo priorisointiin teknisen näkökulman. Työmäärä-arvioiden lisäksi he pystyvät esimerkiksi kertomaan, jos jonkin ominaisuuden toteutus helpottaa ohjelmointityötä jatkossa. Kun kaikki kolme näkökulmaa otetaan huomioon ja tuotteen omistaja, käyttöliittymäsuunnittelijat ja toteutustiimi priorisoivat työlistan yhdessä, saadaan projektin kannalta optimaalisin priorisointi selville luotettavammin.

Kun toteutussprintti aloitetaan sprintin suunnittelukokouksella, käyttöliittymä-suunnittelijoiden tehtävänä on demonstroida käyttöliittymän toimintalogiikka toteutustiimille ja antaa tiimille vastaukset toimintalogiikkaan liittyviin kysymyksiin. Demo tarvitaan, sillä käyttöliittymän toimintalogiikkaa ei dokumentoida

kaikenkattavasti käyttöliittymäkuvaukseen. Käyttöliittymäsuunnittelijat ovat tiimin käytettävissä sprinttien aikana, joten resurssien käyttäminen toimintalogiikan yksityiskohtaiseen dokumentointiin ennen toteutusta ei ole tarpeen. Osa toimintalogiikan kommunikoinnista jätetään tiimin ja käyttöliittymäsuunnittelijoiden yhteydenpidon varaan.

Sprintin aikana tiimin vastuulla on toteutustyö, jonka se voi tehdä niin kuin parhaaksi näkee. Erona alkuperäiseen Scrumiin tiimin ei missään vaiheessa tarvitse ottaa vastuuta käyttöliittymän toimintalogiikan kiinnittämisestä, vaan he voivat keskittyä ohjelmointityöhön. Jos eteen tulee kysymyksiä toimintalogiikasta, voi tiimi vaatia vastaukset käyttöliittymäsuunnittelijoilta. Kun tiimin jäsen saa ohjelmoitua toiminnon valmiiksi, antaa hän sen käyttöliittymäsuunnittelijan hyväksyttäväksi. Käyttöliittymäsuunnittelijat vastaavat toteutustiimin kysymyksiin ja tarkistavat ja hyväksyvät tiimin toteuttaman toiminnallisuuden käyttötilanteita simuloimalla. Jos toteutetuissa interaktioissa ilmenee puutteita, palaa toiminnallisuus ohjelmoijalle korjattavaksi. Sprintin aikana käyttöliittymäsuunnittelijoita ei tarvita kokopäiväisesti ja heillä on tyypillisesti myös muita projekteja työstettävänään. Siten käyttöliittymäsuunnittelijat eivät voi olla tavoitettavissa koko ajan ja vastauksia kysymyksiin tai tarkistusta toteutukselle ei aina saada heti. Vasteaika ei saa olla niin pitkä, että toteutustyö hidastuisi odottamisen takia.

Käyttöliittymäsuunnittelijat eivät pysty osallistumaan kaikkiin Scrumin päivittäisiin seurantakokouksiin, jos heillä on samanaikaisesti muita projekteja. Tämä ei ole ongelma, sillä käyttöliittymäsuunnittelijoiden ei ole tarpeen olla tietoinen jokaisen projektin toteutustyön etenemisestä päivän tarkkuudella. Tärkeämpää on, että käyttöliittymäsuunnittelijat ovat riittävän pienellä vasteajalla tavoitettavissa, kun tiimillä on kysymyksiä.

Scrum-mestarin täytyy olla selvillä GDD:n tuomista muutoksista käytäntöihin. Tärkein näistä on se, että tuotteen työlistaan lisätään vain käyttöliittymään simuloimalla suunniteltuja ominaisuuksia. Esimerkiksi tuotteen omistaja saattaa yrittää lisätä tuotteen työlistaan uusia toimintoja, joita ei ole suunniteltu käyttötilanteita simuloimalla. Jos toiminnot päätyvät sellaisinaan toteutukseen, on

seurauksena hyvin todennäköisesti ongelmallinen käyttöliittymäratkaisu. Scrum-mestari ohjaa uudet toimintopyynnöt käyttöliittymäsuunnittelijoille, jotka selvittävät toimintopyynnön takana olevat käyttötilanteet ja integroivat käyttöliittymään tuen niille.

Kun sprintin katselmoinnissa esitellään toteutettua tuoteinkrementtiä, käyttöliittymäsuunnittelijan tehtävänä on demonstroida toteutetut käyttöliittymäratkaisut käyttötilanteita simuloimalla. Simulointipohjaisen demon perusteella katselmointia seuraavat asiakkaan edustajat näkevät, kuinka järjestelmällä tullaan tekemään töitä. Lisäksi käyttöliittymäsuunnittelija antaa demon seuraajille vastaukset kysymyksiin käyttöliittymän toimintalogiikasta. Jos katselmoinnin seuraajilta tulee ehdotuksia uusista toiminnoista tai muutoksista käyttöliittymään, kirjataan kommentit muistiin ja käyttöliittymäsuunnittelija käsittelee ne myöhemmin. Käsittelyssä selvitetään, onko pyydetylle toiminnolle olemassa käyttötilanteita ja tuki löydetyille tilanteille integroidaan käyttöliittymään GDD-menetelmällä. Muilta osin noudatetaan alkuperäisen Scrumin sprintin katselmointia, jossa tavoitteena on esitellä sprintin aikaansaannokset, jotta katselmoinnin seuraajat voivat arvioida projektin etenemistä ja näkevät, mitä annetuilla resursseilla on saatu aikaan.

Kun järjestelmä on valmis käyttöön otettavaksi, käyttöliittymäsuunnittelija laatii tarpeen vaatiessa käyttötilannepohjaista koulutusta. Koulutus voi olla tarpeen esimerkiksi silloin, kun uusi järjestelmä muuttaa työnkulkuja huomattavasti vanhoihin käytäntöihin nähden. Nähdessään omien työtehtäviensä simulointia järjestelmällä loppukäyttäjät saavat nopeasti käsityksen uudesta työnkulusta.

Taulukossa 5.1 esitetään yhteenveto Scrumin roolien kriittisistä eroista ennen ja jälkeen GDD:n käyttöönoton. Nuolet korostavat keskeiset muuttuneet kohdat vastuissa: tuotteen omistajan ja tiimin aiemmat käyttöliittymän toimintalogiikkaan liittyneet vastuut on siirretty käyttöliittymäsuunnittelijoille. Lihavoinnilla on korostettu muutokset tehtävissä ja vastuissa.

Kun yrityksessä on jo käytössä Scrum ja käyttöliittymäriskit halutaan minimoida ottamalla käyttöön GDD-käyttöliittymäsuunnittelu, tarvitaan vähintään yksi GDD-

Rooli	Vanhat tehtävät ja vastuut	Uudet tehtävät ja vastuut
Uusi rooli: Käyttöliittymä-suunnittelija		Vastaa siitä, että käyttöliittymässä on tarvittava toiminnallisuus ja tietosisältö, sekä hyvä käytettävyys
		Hakee käyttötilanteet kenttätutkimusten avulla
		Kiinnittää käyttöliittymän toimintalogiikan
		Testaa käyttöliittymäratkaisun
Tuotteen omistaja	Vastaa toteutettavasta toiminnallisuudesta ja tietosisällöstä	
	Vastaa projektin liiketoiminnallisesta kannattavuudesta	Vastaa projektin liiketoiminnallisesta kannattavuudesta
	Priorisoi tuotteen työlistan, apuna tiimi	Priorisoi tuotteen työlistan, apuna tiimi **ja käyttöliittymäsuunnittelijat**
Tiimi	Arvioi toteutustyön määrää tuotteen työlistan toiminnallisuuksien perusteella	Arvioi toteutustyön määrää **käyttö-liittymän toimintalogiikan perusteella**
	Vastaa tuotteen työlistan toiminnallisuuksien toteuttamisesta	Vastaa **käyttöliittymään määritellyn toimintalogiikan** toteuttamisesta
	Kiinnittää käyttöliittymän toimintalogiikkaa toteutustyön aikana	
Scrum-mestari	Valvoo Scrumin käytäntöjen ja sääntöjen noudattamista	Valvoo Scrumin käytäntöjen ja sääntöjen, **sekä GDD:n tuomien menettelyjen** noudattamista **(esim. ohjaa toimintopyynnöt käyttöliittymä-suunnittelijoille)**
	Auttaa kaikkia työskentelemään mahdollisimman tehokkaasti poistamalla työntekoa haittaavia esteitä	Auttaa kaikkia työskentelemään mahdollisimman tehokkaasti poistamalla työntekoa haittaavia esteitä

Taulukko 5.1: Scrumin roolien tehtävät ja vastuut ennen ja jälkeen GDD:n käyttöönoton.

menetelmän osaava käyttöliittymäsuunnittelija. Kun Reaktor tarvitsi lisää käyttöliittymäsuunnittelijoita, he päätyivät kouluttamaan GDD-menetelmää olemassa olevalle henkilöstölleen. Muut vaihtoehdot ovat uusien käyttöliittymä-suunnittelijoiden rekrytointi omaan yritykseen tai käyttöliittymäsuunnittelun ostaminen ulkopuolelta konsultointipalveluna. Muiden roolien edustajien nykyiset taidot riittävät, mutta heidät täytyy tehdä tietoisiksi GDD:n tuomista muutoksista.

6 Yhteenveto

Scrum-prosessimallista paikannettiin riskikohtia, jotka saattavat johtaa loppukäyttäjän käyttötilanteita huonosti tukevaan käyttöliittymään. Riskikohtia löytyi kolme: asiakkaan toivomat ominaisuudet päätyvät sellaisinaan ohjelmiston vaatimuksiksi, toimintolistan pohjalta syntyy käytettävyydeltään heikkoja käyttöliittymäratkaisuja ja käyttöliittymän arviointi sprintin katselmoinnissa tuottaa epäluotettavia tuloksia.

Esimerkkitapauksena käytettiin projektia, jossa ohjelmistoyritys Reaktor toteutti ammattikorkeakoulun toiminnansuunnittelujärjestelmän vuosisuunnitteluosion. Scrumin käyttöliittymäriskit onnistuttiin minimoimaan selvittämällä loppukäyttäjien käyttötilanteet kontekstuaalisilla haastatteluilla, suunnittelemalla käyttöliittymä systemaattisesti GDD-menetelmällä ja arvioimalla käyttöliittymää hyödyllisyysläpikäynneillä.

Kun tarvittavat toiminnot, tietosisältö ja toimintalogiikka selvitetään suunnittelemalla käyttöliittymäratkaisu GDD-menetelmällä ennen toteutusvaihetta, osa Scrumin käytännöistä muuttuu. Tavoiteltava ratkaisu on määritelty yksikäsitteisesti käyttöliittymässä, joten toteutustyö nojaa siihen. Tuotteen työlistaan ei enää kirjata asiakkaan toivomia ominaisuuksia, vaan sen sijaan työlista laaditaan käyttöliittymäratkaisun perusteella. Käyttöliittymän pohjalta laadittu työlista auttaa osittamaan ja priorisoimaan toteutustyötä, jonka tavoitteena on toteuttaa käyttöliittymäratkaisun toimintalogiikka.

GDD tuo Scrumiin uuden roolin, käyttöliittymäsuunnittelijan, joka ottaa vastuulleen käyttöliittymän toimintalogiikan. Tämän seurauksena tuotteen omistajan ja tiimin roolit muuttuvat: Vastuu oikean toiminnallisuuden ja tietosisällön määrittelystä on nyt käyttöliittymäsuunnittelijoilla, ei tuotteen omistajalla. Tiimin ei tarvitse ottaa vastuuta käyttöliittymän toimintalogiikan kiinnittämisestä, he voivat vaatia käyttöliittymäsuunnittelijoilta vastaukset toimintalogiikkaan liittyviin kysymyksiin. Tuotteen omistaja lopulta hyväksyy laaditut käyttöliittymäratkaisut, mutta hän ei alkuperäisen Scrumin tavoin määrittele ohjelmiston vaatimuksia kirjaamalla

ominaisuuksia tuotteen työlistaan. Samalla tavalla menetellään toteutusvaiheen aikana tulevien muutosten tai uusien toiminnallisuuksien kanssa: tuotteen omistaja pyytää uutta ominaisuutta käyttöliittymäsuunnittelijoilta, jotka selvittävät pyynnön takana olevan käyttötilanteen ja integroivat toiminnon käyttöliittymään tilannetta simuloimalla. Näin varmistetaan, että käyttöliittymään päätyy hyvä ratkaisu loppukäyttäjän ongelmaan.

Tässä työssä käsitellyssä vuosisuunnitteluprojektissa asiakas toimi passiivisessa roolissa tehdyn työn hyväksyjänä. Tämän työn jatkoksi olisi syytä tarkastella sellaisia GDD:tä ja Scrumia käyttäneitä Reaktorin projekteja, joissa on ollut mukana aktiivinen tuotteen omistaja. Kuinka vaikeaa on ollut saada käyttöliittymäsuunnittelijoiden ja tuotteen omistajan välinen työnjako toimimaan siten, että toimintopyynnöt ohjautuvat käyttöliittymäsuunnittelijoille? Mitä haasteita tässä on ollut ja millä tavalla ongelmat on saatu ratkaistua?

Scrumin käyttöliittymäriskien minimoinnin jälkeen toteutusvaiheeseen jää vielä Scrumista riippumattomia käyttöliittymäriskejä. Vuosisuunnitteluosion toteutusvaiheessa keskeisin riski oli tuoteinkrementtiin päätyneet puutteelliset ja virheelliset käyttöliittymän interaktiot. Riski eliminoitiin hyväksymismenettelyllä, jossa ohjelmoija antaa valmiiksi toteuttamansa toiminnallisuuden käyttöliittymäsuunnittelijan tarkistettavaksi. Hyväksymismenettely edisti tiimin ja käyttöliittymäsuunnittelijoiden välistä kommunikaatiota ja selkeytti työjakoa. Aktiivisempi kommunikaatio ei vain parantanut käyttöliittymätoteutuksen laatua, vaan auttoi myös saamaan kiinni käyttöliittymäsuunnittelussa tapahtuneen inhimillisen virheen. Hyväksymismenettelyllä voidaan minimoida toteutusvaiheen käyttöliittymäriskejä kaikissa projekteissa, joissa toteutustyö nojaa GDD:n tavoin yksikäsitteisesti määriteltyyn toimintalogiikkaan.

Alkuperäisessä Scrumissa liiketoiminnallisesta kannattavuudesta vastaava tuotteen omistaja ja toteutustyöstä vastaava tiimi ottavat varsinaisten tehtäviensä lisäksi vastuulleen myös käyttöliittymän toimintalogiikan. Koska näiltä rooleilta tyypillisesti puuttuu käyttöliittymäsuunnittelun osaaminen, seurauksena on helposti käyttäjän kannalta ongelmallinen järjestelmä. Tässä työssä esitetty uusi roolien tehtävä- ja

vastuunjako siirtää vastuun käyttöliittymän toimintalogiikasta käyttöliittymä-suunnittelijoille, jolloin Scrumin käyttöliittymäriskit voidaan minimoida. Verrattuna vanhoihin rooleihin uudella roolijaolla on huomattavasti paremmat valmiudet tuottaa sellainen ohjelmisto, joka on laadukas myös käyttäjän kannalta.

Lähteet

Abrahamsson02 *Abrahamsson P, Salo O., Ronkainen J., Warsta J.,*
Agile Software Development Methods: Review and Analysis.
VTT Publications 478, 2002.

Abrahamsson04 *Abrahamsson P., Koskela J.,*
Extreme Programming: a Survey of Empirical Data from a Controlled Case Study.
Proceedings of the International Symposium on Empirical Software Engineering, 2004, ISESE '04, s. 73–82.

Bailey93 *Bailey G.,*
Iterative Methodology and Designer Training in Human-Computer Interface Design.
Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems, Amsterdam, The Netherlands, 1993, s. 198–205.

Beck99a *Beck K.,*
Embracing Change with Extreme Programming.
IEEE Computer, Volume 32, October 1999, s. 70–77.

Beck99b *Beck K.,*
Extreme Programming Explained: Embrace Change.
Addison-Wesley, USA, 1999.

Beck00 *Beck K., Fowler M.,*
Planning Extreme Programming.
Addison-Wesley, USA, 2000.

Beyer98 *Beyer H., Holzblatt K.,*
Contextual Design: Defining Customer-Centered Systems.
Morgan Kaufmann Publishers, USA, 1998.

Beyer99 *Beyer H., Holzblatt K.,*
Contextual Design.
ACM interactions, Volume 6, Issue 1, January/February 1999, s. 32–42.

Bias91 *Bias R.,*
Interface-Walkthroughs: Efficient Collaborative Testing.
IEEE Software, Volume 8, No. 5, September 1991, s. 94–95.

Boehm81 *Boehm B.W.,*
Software Engineering Economics.
Prentice-Hall, USA, 1981.

Boehm84 *Boehm B.W., Gray T.E., Seewaldt T.,*
Prototyping vs. Specifying: A Multi-Project Experiment.
Proceedings of the 7th International Conference on Software Engineering, Orlando, Florida, USA, 1984, s. 473–484.

Boehm88 *Boehm B. W.,*
Spiral Model of Software Development and Enhancement.
IEEE Computer, Vol. 21, No. 5, May 1988, s. 61–72.

Boehm91 *Boehm B. W.,*
Software Risk Management: Principles and Practices.
IEEE Software, Volume 8, Issue 1, January 1991, s. 32–41.

Boehm96 *Boehm B. W.,*
Anchoring the Software Process.
IEEE Software, Volume 13, No. 4, July 1996, s. 73–82.

Borälv94 *Borävl E., Göransson B., Olsson E., Sandblad B.,*
Usability and Efficiency. The Helios Approach to Development of User Interfaces.
Computer Methods and Programs in Biomedicine, vol. 45, December 1994, s. 47–64.
PDF:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.5098&rep=rep1&type=pdf (6.5.2010).

Bäumer96 *Bäumer D., Bischofberger W. R., Lichter H., Züllighoven H.,*
User Interface Prototyping—Concepts, Tools, and Experience.
Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, s. 532–541.

Capiluppi07 *Capiluppi A., Fernandez-Ramil J., Higman J., Sharp H. C., Smith N.,*
An Empirical Study of the Evolution of an Agile-Developed Software System.
Proceedings of the 29th International Conference on Software Engineering, 2007, ICSE '07, s. 511–518.

Carroll94 *Carroll J. M.,*
Making Use: A Design Representation.
Communications of the ACM, Vol. 37, No. 12, December 1994, s. 29–35.

Cooper95 *Cooper A.,*
About Face: The Essentials of User Interface Design.
IDG Books Worldwide, Inc, USA 1995.

Cooper03 *Cooper A.,*
About Face 2.0: The Essentials of Interaction Design.
Wiley Publishing, Inc, USA, 2003.

Dagnino02 *Dagnino A.*,
An Evolutionary Lifecycle Model with Agile Practices for Software Development at ABB.
Proceedings of the Eighth IEEE International Conference on Engineering of Complex Computer Systems, 2002, s. 215–223.

Desurvire92 *Desurvire H., Jeffries R.*,
Usability Testing vs. Heuristic Evaluation: Was There a Contest?
ACM SIGCHI Bulletin, Volume 24, Issue 4, October 1992, s. 39–41.

Diaper89 *Diaper D.*,
Task Observation for Human-Computer Interaction.
Teoksessa Diaper D., Task Analysis for Human-Computer Interaction, Ellis Horwood Limited, England, 1989, s. 210–237.

Gilb77 *Gilb T.*,
Software Metrics.
Studentlitteratur, Lund, Sweden, 1977.

Gilb85 *Gilb T.*,
Evolutionary Delivery Versus the "Waterfall Model".
ACM SIGSOFT Software Engineering Notes, Volume 10, Issue 3, July 1985, s. 49–61.

Go04 *Go K., Carroll J. M.*,
The Blind Men and the Elephant: Views of Scenario-Based System Design.
ACM interactions, Volume 11, Issue 6, November/December 2004, s. 44–53.

Gomaa81 *Gomaa H., Scott D. B. H.*,
Prototyping as a Tool in the Specification of User Requirements.
Proceedings of the 5th International Conference on Software Engineering, San Diego, California, United States, 1981, s. 333–342.

Gulliksen95 *Gulliksen J., Sandblad B.*,
Domain-Specific Design of User Interfaces.
International Journal of Human-Computer Interaction, Volume 7, Issue 2, April-June 1995, s. 135–151.
PDF:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.2416&rep=rep1&type=pdf (6.5.2010).

Hackos98 *Hackos J. T., Redish J. C.*
User and Task Analysis for Interface Design.
John Wiley & Sons, USA, 1998.

Highsmith01 *Highsmith J., Cockburn A.,*
Agile Software Development: the Business of Innovation.
IEEE Computer, Volume 34, Issue 9, September 2001, s. 120–127.

Interacta03 *Interacta Design Oy,*
Elektronisen reseptin järjestelmän käyttöliittymäsuunnittelu-projektin dokumentaatio, 2003.

Jensen03 *Jensen B., Zilmer A.,*
Cross-Continent Development Using Scrum and XP.
Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2003, Genova, Italy, May 25–29, Springer-Verlag Berlin / Heidelberg, s. 146–153.

Kniberg08 *Kniberg H., Farhang R.,*
Bootstrapping Scrum and XP under Crisis. A Story from the Trenches.
Proceedings of the Agile 2008 Conference, 4.-8. August 2008, Toronto, Canada, s. 436–444.

Laakso04a *Laakso S.A., Laakso K-P.,*
Hyvän käyttöliittymän varmistaminen GUIDe-prosessimallilla.
Julkaisematon artikkeli, Helsingin yliopisto, tietojenkäsittelytieteen laitos, 2004.
PDF: http://www.cs.helsinki.fi/u/salaakso/papers/GUIDe-suomeksi.pdf (6.5.2010).

Laakso04b *Laakso S.A., Laakso K.-P., Latva-Koivisto A.,*
Käyttöliittymien suunnittelumallit.
Käyttöliittymät II –kurssin luentomoniste, Helsingin yliopisto, tietojenkäsittelytieteen laitos, 2004.
PDF:http://www.cs.helsinki.fi/u/salaakso/kl2-2004/Kayttoliittymat2-Luento9-24.11.2004-Sari-A-Laakso.pdf (6.5.2010).

Laakso05 *Laakso S.A.,*
Käyttöliittymät 2005.
Luentomoniste, Helsingin yliopisto, tietojenkäsittelytieteen laitos, sarja D 424, 2005.
PDF: http://www.cs.helsinki.fi/u/salaakso/papers/Kayttoliittymat-opetusmoniste-2005.pdf (6.5.2010).

Laakso06a *Laakso S.A., Latva-Koivisto A.,*
Käyttöliittymät 2006.
Luentomoniste, Helsingin yliopisto, tietojenkäsittelytieteen laitos, sarja D-2006-1, 2006.
PDF: http://www.cs.helsinki.fi/u/salaakso/papers/Kayttoliittymat-opetusmoniste-2006.pdf (6.5.2010).

Laakso06b *Laakso S.A.*,
Käyttöliittymälähtöinen vaatimusmäärittely.
Systeemityö, Nro 2, 2006, s.19–21.
PDF: http://www.cs.helsinki.fi/u/salaakso/papers/SYTYKE-Kalilahtoinen-vaatimusmaarittely.pdf (6.5.2010).

Laakso06c *Laakso S.A., Latva-Koivisto A.*,
Käyttöliittymien arviointimenetelmät.
Käyttöliittymät II –kurssin luentomoniste, Helsingin yliopisto, tietojenkäsittelytieteen laitos, 2006.
PDF:http://www.cs.helsinki.fi/u/salaakso/kl2-2006/Kayttoliittymat2-Luento7-Simulointitestaus-hyodyllisyyslapik-ja-muut.pdf (6.5.2010).

Laakso06d *Laakso S.A., Latva-Koivisto A.*,
Skenaariot, käyttötapaukset ja päätöksentekokohdat.
Käyttöliittymät II –kurssin luentomoniste, Helsingin yliopisto, tietojenkäsittelytieteen laitos, 2006.
HTML:http://www.cs.helsinki.fi/u/salaakso/kl2-2006/Kayttoliittymat2-Skenaariot-kayttotapaukset-paatoksenteko.html (6.5.2010).

Laakso07 *Laakso K.*,
Kokemuksia GUIDe-käyttöliittymäsuunnittelun ja Scrum-menetelmän yhdistämisestä,
Systeemityö, Nro 4, 2007, s.16–18.
PDF:http://www.pcuf.fi/sytyke/lehti/kirj/st20074/ST074-16A.pdf (6.5.2010).

Larman03 *Larman C., Basili V.R.*,
Iterative and Incremental Development: A Brief History.
IEEE Computer, Volume 36, Issue 6, June 2003, s. 47–56.

Larman07 *Larman C.*,
Agile & Iterative Development: A Manager's Guide.
Addison-Wesley, USA, 2007.

Lauesen01 *Lauesen S., Harning M.B.*,
Virtual Windows: Linking User Tasks, Data Models, and Interface Design.
IEEE Software, Volume 18, Issue 4, July/August 2001, s. 67–75.

Lauesen05 *Lauesen S.*,
User Interface Design: a Software Engineering Perspective.
Addison Wesley, UK, 2005.

Lewis97 *Lewis C., Wharton C.*,
Cognitive walkthroughs.
Teoksessa Helander M., Landauer T., Pradhu P. (toim.), Handbook of Human-Computer Interaction. Elsevier Science B.V., 1997, s. 717–732.

MacCormack01 *MacCormack A.,*
Product-Development Practices That Work: How Internet Companies Build Software.
MIT Sloan management review, Vol. 42, No. 2, 2001, s. 75–84.

Mann05 *Mann C., Maurer F.,*
A Case Study on the Impact of Scrum on Overtime and Customer Satisfaction.
Proceedings of the Agile 2005 Conference, Denver, Colorado, USA, July 2005, s. 70–79.

Mar02 *Mar K., Schwaber K.,*
Scrum with XP.
InformIT (www.informit.com), March 22, 2002.
HTML: http://www.informit.com/articles/article.aspx?p=26057 (6.5.2010).

Martin06 *Martin A., Noble J., Biddle R.,*
Programmers Are from Mars, Customers are from Venus: A Practical Guide for Customers on XP Projects .
Proceedings of the 2006 conference on Pattern languages of programs, ACM International Conference Proceeding Series, Article No. 20, 2006.

Maurer02 *Maurer F., Martel S.,*
Extreme Programming. Rapid Development for Web-Based Applications.
IEEE Internet Computing, Volume 6, Issue 1, January/February 2002, s. 86–90.

McCracken81 *McCracken D. D., Jackson M. A.,*
Life Cycle Concept Considered Harmful.
ACM SIGSOFT Software Engineering Notes, Volume 7, Issue 2, April 1982, s. 29–32.

Mills76 *Mills H. D.,*
Software Development.
IEEE Transactions on Software Engineering, Volume SE-2, Issue 4, December 1976, s. 265–273.

Moore07 *Moore R., Reff K., Graham J., Hackerson B.,*
Scrum at a Fortune 500 Manufacturing Company.
Proceedings of the Agile 2007 Conference, Washington D.C., USA, August 2007, s. 175–180.

Moser08 *Moser R., Abrahamsson P., Pedrycz W., Sillitti A., Succi G.,*
A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team.
Balancing Agility and Formalism in Software Engineering, Lecture Notes in Computer Science, Volume 5082/2008, Springer Berlin / Heidelberg, 2008, s. 252–266.

Mäkinen07 *Mäkinen V.-M.,*
Reaktorin sisäisen koulutustilaisuuden materiaali, tammikuu 2007.

Neill03 *Neill C. J., Laplante P.A.,*
Requirements Engineering: The State of the Practice.
IEEE Software, Volume 20, Issue 6, November/December 2003, s. 40–45.

Nielsen93 *Nielsen J.,*
Usability Engineering.
Academic Press, New York, 1993.

Nielsen01 *Nielsen J.,*
First Rule of Usability? Don't Listen to Users.
Alertbox: Jakob Nielsen's Newsletter on Web Usability, August 5, 2001. HTML:http://www.useit.com/alertbox/20010805.html (6.5.2010).

Nielsen03 *Nielsen J.,*
Paper Prototyping: Getting User Data Before You Code.
Alertbox: Jakob Nielsen's Newsletter on Web Usability, April 14, 2003. HTML:http://www.useit.com/alertbox/20030414.html (6.5.2010).

Norman88 *Norman D.A.,*
The Design of Everyday Things.
Basic Books, New York, USA, 1988.

Norman05 *Norman D.A.,*
Human-centered Design Considered Harmful.
ACM interactions, Volume 12, Issue 4, July/August 2005, s. 14–19.

Orr04 *Orr K.,*
Agile Requirements: Opportunity or Oxymoron?
IEEE Software, May/June 2004, Volume 21, Issue 3, s. 71–73.

Parnas94 *Parnas D. L.,*
Software Aging.
Proceedings of the International Conference on Software Engineering, Sorrento, Italy, 1994, s. 279–287.

Reaktor06 *Reaktor Innovations Oy,*
Toisu-järjestelmän vuosisuunnitteluosion kehitysprojektin tuottamat käyttöliittymäkuvaukset, tuotteen työlistat ja sprintin työlistat, 2006–2007.

Rettig94 *Rettig M.*,
Prototyping for Tiny Fingers.
Communications of the ACM, Volume 37, Issue 4, April 1994, s. 21–27.

Robertson02 *Robertson J.*,
Eureka! Why Analysts Should Invent Requirements.
IEEE Software, Volume 19, Issue 4, July/August 2002, s. 20–22.

Royce70 *Royce W. W.*,
Managing the Development of Large Software Systems: Concepts and Techniques.
IEEE Western Conference (Wescon), 1970, s. 1–9.

Salo08 *Salo O., Abrahamsson P.*,
Agile Methods in European Embedded Software Development Organisations: a Survey on the Actual Use and Usefulness of Extreme Programming and Scrum.
IET Software, Volume 2, Issue 1, February 2008, s. 58–64.

Schwaber01 *Schwaber K., Beedle M.*,
Agile Software Development with Scrum.
Prentice Hall, USA, 2001.

Schwaber04 *Schwaber K.*,
Agile Project Management with Scrum.
Microsoft Press, USA, 2004.

Singh08 *Singh M.*,
U-SCRUM: An Agile Methodology for Promoting Usability.
Proceedings of the Agile 2008 Conference, 4.-8. August 2008, Toronto, Canada, IEEE Computer Society, s. 555–560.

Sommerville07 *Sommerville I.*,
Software Engineering (Eighth Edition).
Addison Wesley, UK, 2007.

Stadia04 *Helsingin ammattikorkeakoulu Stadia,*
Tarjouspyyntö ammattikorkeakoulun toiminnansuunnittelu-järjestelmästä, liite 4: Järjestelmän käyttöliittymän kuvaus, rautalankamalli, 5.11.2004.

Sutherland01 *Sutherland J.,*
Agile Can Scale: Inventing and Reinventing SCRUM in Five Companies.
Cutter IT Journal, Volume 14, No. 12, December 2001, s. 5–11.

Sutherland07 *Sutherland J., Schwaber K.,*
The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process.
Kirjan luonnos, 22.4.2007.
PDF: http://www.crisp.se/scrum/books/ScrumPapers20070424.pdf (6.5.2010).

Takeuchi86 *Takeuchi H., Nonaka I.,*
The New Product Development Game.
Harvard Business Review, Volume 64, 1986, s. 137–146.

Virzi96 *Virzi R. A., Sokolov J. L., Karis D.,*
Usability Problem Identification Using both Low- and High-Fidelity Prototypes.
Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Common Ground, Vancouver, British Columbia, Canada, 1996, s. 236–243.