Antti Kohtamäki

OHJELMISTON DOKUMENTOINNIN TUKITYÖKALU

Tietotekniikan koulutusohjelma

2016

Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

OHJELMISTON DOKUMENTOINNIN TUKITYÖKALU

Kohtamäki, Antti
Satakunnan ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Huhtikuu 2016
Ohjaaja: Aromaa, Juha DI
Sivumäärä: 36
Liitteitä: -

Asiasanat: dokumentointi, ohjelmistotuotanto, automaatio

Opinnäytetyön aiheena oli tukityökalun toteuttaminen suomalaiselle ohjelmistoalan yritykselle matkapuhelinverkon tukiasemaohjaimen ohjelmiston dokumentoinnin helpottamiseksi. Samanlaista tai vastaavaa ohjelmaa yrityksessä ei ole saatavilla ja sen kehittäminen nähtiin tarpeellisena ohjelmiston koodimäärän kasvaessa ja dokumentointiin käytetyn työmäärän lisääntyessä. Dokumentoinnin työlästä luonnetta haluttiin lieventää työkalulla, joka tuottaisi automaattisesti osia TNSDL -koodin dokumentoinnista sekä toimisi ohjelmoijalle tukena ohjelmistokehityksessä.

Opinnäytetyössä perehdyttiin yleisellä tasolla ohjelmistotuotantoon, testaukseen ja automaatioon, sekä läpikäytiin työkalun spesifikaatiot.

Tukityökalu tehtiin yrityksen tiloissa ja haluttuja ominaisuuksia kartoitettiin ohjelmoijilta. Tukityökalu toteutettiin käyttäen C++ -, JavaScript (jQuery) -, HTML5 -, sekä CSS -ohjelmointi -ja kuvauskieliä. Työkalun käyttöliittymä toteutettiin käyttäen Windows -lomakkeita sekä selainpohjaista ratkaisua.

SUPPORT TOOL FOR SOFTWARE DOCUMENTATION

Kohtamäki, Antti
Satakunta University of Applied Sciences
Degree Programme in Information Technology
April 2016
Supervisor: Aromaa, Juha M.Sc.
Number of pages: 36
Appendices: -

Keywords: documentation, software development, automation

The purpose of this thesis was to create a support tool for a Finnish software industry company mobile network base station controller to ease software documentation. There is no equal or similar program available in the company and its development was seen necessary as both the amount of code in the software and the workload needed for documentation increases. The support tool was wanted to mitigate the arduous nature of documentation by automatically producing parts of the TNSDL code documentation and act as support for the programmer in software development.

Software development, testing and automation are introduced generally, and the specifications of the tool are presented in the thesis.

The support tool was created in company premises and the desired features were inquired from the programmers. The support tool was created using C++, JavaScript (jQuery), HTML5, and CSS programming and markup languages. The user interface was created using Windows forms and browser based solution.

SISÄLLYS

LYHENTEET JA TERMIT

CSS – Cascading Style Sheets

HTML – HyperText Markup Language

IEC – International Electro technical Commission

IEEE – Institute of Electrical and Electronics Engineers

ISTQB – International Software Testing Qualifications Board

ISO – International Organization for Standardization

Komponentti – itsenäinen, palveluja tarjoava ohjelmistoyksikkö

SPICE – Software Process Improvement and Capability dEtermination

SWEBOK – Software Engineering Body of Knowledge

TMMI – Test Maturity Model Integration

TNSDL – TeleNokia Specification and Description Language

XP – eXtreme Programming

JOHDANTO

Ohjelmiston koodimäärän kasvaessa kokonaisuuden hahmottaminen hankaloituu. Koodin rakennetta on vaikeampi ymmärtää koodin hajautuessa useaan eri tiedostoon. Koodimäärän kasvaessa myös ajankohtaisen dokumentaation ylläpito käy työlääksi. Koodivirheiden määrä lisääntyy ja sekä koodia, että dokumentaatiota on alati päivitettävä.

Ohjelmoijalla on siis jatkuva tarve tarkastella ja muokata koodia, sekä koodin dokumentaatiota. Lisäksi ohjelmoijan on hahmotettava koodin rakenne, jotta hän ei muutoksillaan aiheuttaisi lisävirheitä.

Näistä asioista johtuen haluttiin opinnäytetyössä kehittää tukityökalu ohjelmoijan tueksi ohjelmointikehityksessä ja dokumentaation ylläpidossa. Tukityökalu loisi koodista HTML -dokumentin, joka toimisi koodin ja dokumentaation välikappaleena, linkittämällä kooditiedostot ja luomalla automaattisesti osia dokumentaatiosta.

1 OHJELMISTOTUOTANTO

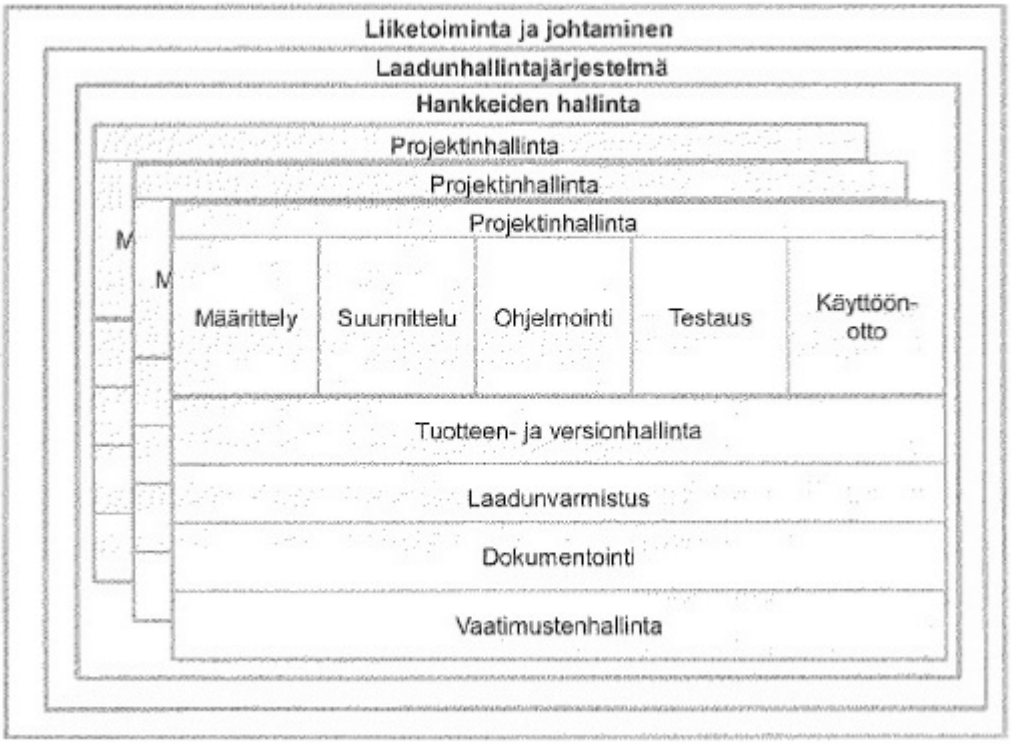
Käsite ohjelmistotuotanto tai “Software Engineering” tarkoittaa ohjelmistojen rakentamisessa yleisesti käytettyjä tekniikoita, työkaluja, menettelytapoja ja periaatteita. Ohjelmistotuotannon lähtökohtana eivät ole muissa insinööritieteissä yleisesti käytetyt matemaattiset menetelmät, vaan pikemminkin se koostuu joukosta hyväksi havaittuja toimintatapoja ja periaatteita. (Haikala & Mikkonen 2011, 11)

Ohjelmisto tuotetaan tyypillisesti projektina, jossa toteutetaan asiakkaalta saatuihin vaatimuksiin perustuen haluttu ohjelmisto. Projektille on määritelty selkeä sisältö ja tavoitteet. Tavoitteiden saavuttamiseksi projektille on annettu resursseja, esimerkiksi henkilöitä, alihankintaan käytettävää rahaa tai jotain muuta, jonka avulla projektin läpivieminen on mahdollista. Lisäksi projektilla on oltava aikataulu, joka luo puitteet projektin kestolle ja jonka avulla hallitaan projektin etenemistä. (Haikala & Mikkonen 2011, 21)

Ohjelmistotuotannon tarkoituksena on siis tuottaa ohjelmia, jotka täyttävät asiakkaan asettamat kohtuulliset odotukset niin, että kehityskustannukset ja aikataulu ovat ennustettavissa riittävän tarkasti. (Haikala & Märijärvi 2006, 16)

1.1 Ohjelmistotuotannon osa-alueet

Ohjelmistokehitykseen liittyy paljon muutakin kuin itse ohjelmointi. Ohjelmistotuotanto koostuu ohjelmistotyön vaiheista, kuten määrittely, suunnittelu, toteutus, testaus ja laadunvarmistus, sekä niitä ympäröivistä tuotantoprosessin osa-alueista, kuten laatujärjestelmä, tuotteenhallinta, projektinhallinta ja dokumentointi (Kuva 1). (Haikala & Mikkonen 2011, 12)



Kuva 1. Ohjelmistotuotannon osa-alueet (Haikala & Mikkonen 2011, 29)

1.1.1 Määrittely

Määrittelyllä (specification, functional specification) eli speksauksella tarkoitetaan projektin asiakasvaatimusten ja niistä johdettujen ohjelmiston toimintojen dokumentointia asiakasnäkökulmasta. Määrittelydokumentissa eli speksissä dokumentoidaan, mitä asiakkaalle tarkkaan ottaen ollaan tekemässä. Speksin asettamat rajat ja vaatimukset ovat suunnitteluvaiheen ohjenuoria. (Haikala & Mikkonen 2011, 30)

1.1.2 Suunnittelu

Suunnittelulla (architectural design, design) tarkoitetaan määrittelyn mukaisen ohjelmiston teknistä suunnittelua. Arkkitehtuurisuunnittelun tehtävä on määritellä ohjelmiston komponentit, työnjako niiden välillä, vastuut ja tehtävät sekä komponenttien väliset suhteet. (Haikala & Mikkonen 2011, 178)

1.1.3 Toteutus

Projektin toteutusvaiheessa (implementation) ohjelmiston koodi kirjoitetaan suunnitteluvaiheessa määriteltyjen ohjenuorien mukaisesti. Ohjelmoinnin lisäksi toteutusvaiheeseen kuuluu usein myös suunnittelua ja testausta. (Haikala & Märijärvi 2006, 41)

1.1.4 Testaus

Testauksella (testing) tarkoitetaan suunnitelmallista virheiden etsimistä ja se käsittää kaikki menetelmät, joilla pyritään mittaamaan ja parantamaan ohjelman laatua. Testauksen työvaiheita ovat testauksen suunnittelu (testaussuunnitelma ja testitapaukset), testiympäristön luonti, testin suorittaminen ja tulosten tarkastelu (testiraportti). Testaukseen, virheiden jäljitykseen ja korjaukseen (debugging) kuluu enemmistö projektin resursseista. (Haikala & Märijärvi 2006, 283–284)

1.1.5 Dokumentointi

Dokumentointi (documentation) on osa lähes kaikkia ohjelmistotuotannon osa-alueita ja on tärkeä osa projektinhallintaa. Ohjelmiston keskeisin dokumentti on sen arkkitehtuurin kuvaus. Se sisältää suunnittelijoiden näkemyksen ja järjestelmän perusrakenteen. Se toimii myös järjestelmän ylläpito-ohjeena sekä perehdytysmateriaalina. (Haikala & Mikkonen 2011, 194)

1.1.6 Käyttöönotto & Ylläpito

Projektin käyttöönottovaiheessa (commissioning & maintenance) ohjelmisto luovutetaan asiakkaalle ja asiakasta tuetaan, kunnes ohjelmisto on aktiivikäytössä. Sen jälkeen alkaa ohjelmiston ylläpitovaihe, jossa asiakasta neuvotaan mahdollisissa ongelmatilanteissa ja korjataan ilmenneitä ongelmia ja päivitetään ohjelmistoa. (Haikala & Märijärvi 2006, 41)

1.1.7 Tuotteenhallinta

Tuotteenhallinta (configuration management) pitää kirjaa ohjelmistotuotteen tilasta, mitä komponentteja ja niiden eri versioista on olemassa, mistä versiosta viimeisin julkistus (release) on koostettu (build). (Haikala & Mikkonen 2011, 30)

1.1.8 Laadunvarmistus

Laadunvarmistuksella (quality assurance) viitataan menetelmiin, joilla pyritään varmistumaan tuotteen ja käytetyn prosessin laadusta. Siihen sisältyy mm. tuotteen testaus, arviointi sekä tarkastus. Laadukas prosessi tuottaa kustannustehokkaasti halutunlaisen tuotteen. (Haikala & Märijärvi 2006, 267–268)

1.1.9 Vaatimustenhallinta

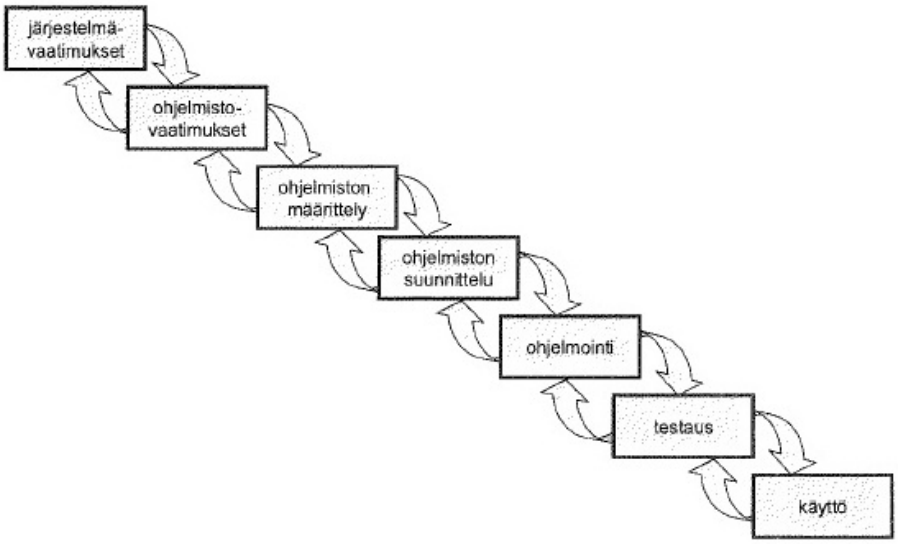
Vaatimustenhallinta (requirements management) käsittää keinot, joilla varmistetaan, että lopputuote vastaa asiakkaan vaatimuksia. Se sisältää myös menettelyt vaatimusmuutosten käsittelyyn. (Haikala & Märijärvi 2006, 91)

1.2 Ohjelmistotuotannon projektimalleja

Vuosien varrella on kehitetty monia erilaisia projektimalleja ja niiden ympärille on kasvanut oma liiketoimintansa. Malleja muokataan asiakkaan projektiin soveltuvaksi ja niiden käyttöä varten järjestetään koulutuksia. Eri projektit saattavat käyttää eri malleja, mutta perusperiaatteiltaan ne ovat kuitenkin hyvin samanlaisia.

1.2.1 Vesiputousmalli

Vuonna 1970 amerikkalainen ohjelmistotuotannon pioneeri Winston W. Royce julkaisi yhden alan eniten viitatuista ja samalla myös eniten väärinymmärretyistä artikkeleista: Managing the Development of Large Software Systems. Sen keskeisin sisältö oli vesiputousmalli, jonka yksi tärkein, mutta unohdettu ominaisuus oli taaksepäin iterointi (Kuva 2).



Kuva 2. Vesiputousmalli (Haikala & Mikkonen 2011, 37)

Vesiputousmalli yksinkertaistui usein malliksi, josta iterointi puuttui ja näin syntyi kuva, että vesiputousmallia ei käytännössä ole mahdollista noudattaa. Sallimalla kuitenkin iteroinnin ja vaiheiden käynnistämisen ennen edellisen loppua, saadaan moneen tilanteeseen sopiva toimintamalli. Projektimalleja tarkasteltaessa löytyy niiden sisältä yleensä vesiputousmallin alkuperäinen idea jollakin tavoin toteutettuna. Esimerkiksi Scrum-malli voidaan nähdä sarjana toisiaan seuraavia lyhyitä vesiputouksia. (Haikala & Mikkonen 2011, 37)

1.2.2 Protoilu

Protoilulla tarkoitetaan vaillinaisen prototyypin rakentamista ohjelman ominaisuuksien tutkimista varten. Protoilua on kahdenlaista: evoluutioprototyyppi (evolutionary prototype) ja poisheitettävä (throw-away prototype). Evoluutioprototyyppi kehitetään vaiheittain valmiiksi tuotteeksi ja poisheitettävää prototyyppiä käytetään järjestelmän mallintamiseen, minkä jälkeen tuotteen kehittäminen aloitetaan alusta. Lähestymistapojen välimuodot ovat mahdollisia, jolloin jotkut osat prototyypistä hyödynnetään ja osa korvataan uudella toteutuksella.

Evoluutioprototyyppiä käytettäessä ohjelman prosesseja ja moduuleja toteutetaan ensin prototyyppeinä. Tämän jälkeen protoja korvataan todellisilla toteutuksilla. Tällaiset prototyypit sisältävät yleensä paljon uudelleenkäytettävää koodia. Poisheitettäviä prototyyppejä käytetään yleensä käyttöliittymän suunnittelussa (Näyttöjen ulkonäkö, navigointi, yms.).

Prototyyppien vaarana on, että ne jäävät osaksi lopullista järjestelmää. Prototyyppikomponentti saattaa toimia oikein, mutta olla toteutukseltaan erittäin hidas. Projektin loppukiireessä tällainen komponentti voi jäädä korvaamatta oikealla toteutuksella tai jäädä jopa kokonaan huomaamatta. Prototyypin tulisi viestiä keskeneräisyydestä, sillä hyvinkin toimiva prototyyppi ei ole vielä valmis tuote. (Haikala & Mikkonen 2011, 38–39)

1.2.3 Agile

Ketterät (agile) menetelmät kehitettiin Yhdysvalloissa raskaiden suunnitelmaohjattujen (plan-driven) kehitysprosessien vastapainoksi. Agilen ydin on sen arvoissa (Agile Manifesto). Se painottaa tyytyväisen asiakkaan ja toimivan ohjelmiston tärkeyttä eikä niinkään käytettyä prosessia, työkaluja tai dokumentointia, vaikka toteaakin ne tärkeiksi ohjelmistokehitystyön kannalta. (Haikala & Mikkonen 2011, 43–44)

Taulukko 1. Agilen arvot (Haikala & Mikkonen 2011, 45)

	Periaate
1	Asiakas on pidettävä tyytyväisenä toimittamalla tälle projektin alusta asti tasaisella tahdilla toimivia ohjelmistoversioita.
2	Vaatimuksien muuttuminen tulee hyväksyä projektin aikana, myös myöhäisissä kehitysvaiheissa.
3	Toimivien asiakasversioiden julkaisuväli voi vaihdella muutamasta viikosta muutamaan kuukauteen (mieluummin viikkoja kuin kuukausia).
4	Liiketoiminta- ja kehitysrooleissa olevien työntekijöiden tulee työskennellä yhdessä päivittäin koko projektin ajan.
5	Projektit tulee rakentaa motivoituneiden yksilöiden ympärille. Heille tulee antaa ympäristö ja tuki, jota he tarvitsevat työn tekemiseksi. Luota siihen, että he saavat työn tehtyä.
6	Kaikkein tehokkain kommunikointikeino sekä kehitystiimin ja ulkomaailman välillä, että itse kehitystiimissä on keskustelu kasvokkain.
7	Toimiva ohjelmisto on projektin edistymisen tärkein mittari.
8	Ketterät menetelmät edistävät tasaista kehitystahtia. Projektin parissa työskentelevien henkilöiden tulisi kyetä pitämään työtahti ja -kuorma mahdollisimman tasaisena, eikä ylitöitä kuulu tehdä.
9	Huomion jatkuva kiinnittäminen tekniseen erinomaisuuteen ja hyvään suunnitteluun tehostavat ketteryyttä.
10	Asiat pitäisi aina pyrkiä tekemään mahdollisimman yksinkertaisella tavalla: minimoidaan vähemmän tärkeät tehtävät.
11	Parhaat arkkitehtuurit, vaatimukset ja suunnitelmat syntyvät itseohjautuvasti organisoituneissa tiimeissä.
12	Tiimin tulee selvittää säännöllisin väliajoin, miten se voisi tulla aikaisempaa tehokkaammaksi ja hienosäätää käytettävää työtapaa ja prosessia sen mukaisesti.

Taulukon 1 arvot ovat tarkoituksenmukaisia, mutta niiden lähtökohtana on kuitenkin ajatus pienestä asiakasprojektista. Laajoihin projekteihin näitä periaatteita voi olla vaikea tai mahdoton soveltaa. Esimerkiksi asiakkaan jatkuva läsnäolo ei yleensä ole mahdollista (periaatteet 4 ja 5) ja kasvokkain kommunikointi maantieteellisesti hajautetuissa hankkeissa on haasteellista (periaate 6).

Ketteryys on levinnyt laajalle ja saanut paljon huomiota, mutta sen menetelmiä ei enää käytetä sellaisenaan. Agilen ideoita, esimerkiksi XP:n (eXtreme Programming) jatkuva integrointi ja lyhyet iteraatiot, sovelletaan kuitenkin osana Scrumia. (Haikala & Mikkonen 2011, 45–46)

1.2.4 Scrum

Viime vuosina yleistynyt ketterä menetelmä on Japanista peräisin oleva Scrum. Scrumin merkittävä etu on sen yksinkertaisuus: perusperiaatteet on helppo selittää. Scrum on lähinnä projektin toteutusvaiheeseen tarkoitettu tapa organisoida projektin iteraatiot. Scrum ottaa kantaa vain osaan ohjelmiston elinkaaren tehtävistä, eikä se ota kantaa käytettyihin kehitysmenetelmiin tai työkaluihin.

Scrumissa on kolme roolia: tuotteen omistaja (product owner), Scrum-mestari (Scrum Master) ja tiimi. Tuotteen omistajan toimenkuva muistuttaa tuotepäällikköä (eli tuotteen omistajaa), Scrum-mestari projektipäällikköä ja tiimi projektiryhmää.

Tuotteen omistaja vastaa projektin taloudellisesta tuloksesta. Hän toimii projektin rajapintana sidosryhmiin, joilta hän kerää järjestelmän vaatimukset tuotteen työlistalle (product backlog), jonka alkioita (item) hän ylläpitää prioriteettijärjestyksessä. Työlista-alkiolla on alustava aika-arvio sekä arvio sen liiketoiminta-arvosta. Työlistan alkiot voivat olla esimerkiksi tuotteen ominaisuuksia, vaatimuksia tai virheraportteja. Vaatimustenhallinta on tuotteen omistajan vastuulle.

Scrum-mestari vastaa, että Scrum-prosessia noudatetaan, hän toimii tiimin ja tuotteenomistajan valmentajana. Hän vastaa pyrähdyksen tuloksesta ja varmistaa, että tehtävää ei merkitä valmiiksi ennen kuin sen valmistumisen toteamiseen määritellyt ehdot (definition-of-done, DoD) on täytetty, esimerkiksi koodi kirjoitettu, testitapaukset olemassa ja ajettu onnistuneesti sekä dokumentaatio päivitetty. Scrum-mestari vastaa myös tiimin hyvinvoinnista ja tiimin työtä haittaavien esteiden (impediment), jopa sopimattoman jäsenen poistamisesta.

Tiimin optimaalinen koko on noin 7 henkilöä, jotka saisivat olla taustoiltaan erilaisia (cross functional team). Tiimi on itseorganisoituva ja päättää itse omista työskentelykäytännöistään. Tiimi vastaa pyrähdyksen työlistan paloittelusta tehtäviksi ja jakaa ne keskenään. Tiimin tulisi olla samassa työtilassa, johon voidaan myös sijoittaa tehtävätaulu, josta tehtävät ja niiden tilat ovat kaikkien nähtävissä. Tiimi ylläpitää taulua Scrum-mestarin kanssa. Päivittäiset Scrum-palaverit (daily scrum) voidaan pitää seisaallaan taulun ääressä. (Haikala & Mikkonen 2011, 46–49)

Scrumissa projekti etenee pyrähdyksinä (sprint), joiden pituus on 30 kalenteripäivää. Jokainen pyrähdys alkaa pyrähdyksen suunnittelukokouksella (sprint planning meeting). Kokoukseen osallistuvat tuotteen omistaja, Scrum-mestari ja tiimi. Kokouksessa tuotteen omistaja esittelee tuotteen työlistan, jonka jälkeen sovitaan, mitkä tuotteen tehtävälistan alkiot otetaan mukaan seuraavaan pyrähdykseen, ts. tiimi päättää, mihin se voi sitoutua.

Tehtävälistan alkiot pilkotaan tehtäviin (task), joiden kesto on tyypillisesti 4-16 tuntia. Muutaman pyrähdyksen jälkeen tiimi oppii kuinka monta tehtävää voidaan laittaa yhteen pyrähdykseen. Tätä sanotaan tiimin nopeudeksi (velocity). Pyrähdyksen seurantaan käytetään edistymiskäyrää (sprint burndown chart), jota päivitettäessä nähdään miten pyrähdyksen työmäärä kehittyy.

Pyrähdyksen aikana pyrähdyksen tehtävälistaa ei muuteta. Tämä on selkeä muutos perinteisiin tapoihin, sillä ennen jouduttiin varautumaan lähes päivittäisiin vaatimusmuutoksiin. Nyt käytettävissä on ainakin pyrähdyksen mittainen vaatimusmuutoksilta rauhoitettu aika. Pyrähdyksen aikana pidetään joka päivä noin 15 minuutin päivän Scrum -kokous (daily scrum). Kokoukseen osallistuvat Scrum-mestari ja tiimi. Jokainen tiimin jäsen vastaa kolmeen kysymykseen:

- Mitä olet tehnyt edellisen kokouksen jälkeen?
- Mitä aiot tehdä seuraavaksi?
- Onko tiedossasi esteitä, jotka hidastavat työtäsi?

Pyrähdyksen päätyttyä pidetään katselmointikokous, johon osallistuvat tiimi, Scrum-mestari, tuotteen omistaja ja mahdollisesti sidosryhmien edustajia (asiakkaita). Tiimi demonstroi pyrähdyksen tulokset ja kerää palautteen.

Pyrähdyksen arviointipalaveriin (retrospective) osallistuvat tiimi, Scrum-mestari ja tuotteen omistaja. Palaverissa arvioidaan pyrähdyksen onnistumista ja mietitään toimintatapoja kehittämistä. Jokainen osallistuja valmistautuu vastaamaan kahteen kysymykseen: mikä meni hyvin ja mitä pitäisi parantaa seuraavassa pyrähdyksessä. (Haikala & Mikkonen 2011, 49–51)

Käytännössä Scrumista on käytössä omiin tarkoituksiin sovellettu versio Scrumista (Scrumbut). Tavallisia poikkemia ovat päivittäisen Scrum-palaverin pitämisen vain kerran viikossa, lyhyet 2 viikon pyrähdykset tai puutteellinen tuotteen työlista. Scrumin omaksuminen edellyttää yritysjohdon sitoutumista ajan ja rahoituksen muodossa sekä käytännön koulutusta. Tyypillinen käyttöönotto vaatii noin puoli vuotta aikaa, jonka aikana tuotekehityksen vauhti voi aluksi jopa hidastua. Yksi vaikeimmista asioista on pyrähdyksen työlistan pilkkominen riittävän pieniin tehtäviin.

Tuotteen omistajan rooli on tuotteen kannalta kriittisin, hän vastaa vaatimustenhallinnasta (tuotteen työlistan ylläpidosta) ja etenemissuunnitelmasta. Hän huolehtii, että sidosryhmien näkemykset otetaan huomioon vaatimustenhallinnassa. Noin 60 % tuotteen virheistä johtuu vaatimuksissa olevista virheistä, joten ainakaan laajoissa projekteissa pelkkä tuotteen työlistan ylläpito ei riitä, vaan tarvitaan myös vaatimustenhallinnan perinteisiä menetelmiä.

Scrum-mestari valvoo, että prosessia noudatetaan. Tärkeintä on, että kaikki tehtävät, jotka liittyvät ohjelmiston valmiiksi saamiseen on tehty kunnolla, testaus on riittävän kattava ja dokumentaatio on ajan tasalla. Muuten projektille syntyy teknistä velkaa, joka johtaa projektin loppuvaiheissa aikataulutusongelmiin.

Yksinkertaisimmillaan Scrum on systemaattinen tapa ohjata projektia: se ei ole edes kovin ketterä, vaan enemmänkin korostetun iteratiivinen ja kattaa vain osan projektin hallintaan liittyvistä asioista. (Haikala & Mikkonen 2011, 52–54)

1.2.5 Lean & Kanban

Scrumin lisäksi Japanista on saatu Lean (manufacturing) -ajattelutapa, jota käytetään apuna prosessia kehitettäessä, sekä yksinkertainen työnhallintatapa Kanban.

Leanin kaksi keskeisintä ajatusta ovat:

- Eliminoi hukka (eliminate waste): hukkaa on kaikki, mikä ei tuota asiakkaalle lisäarvoa. Myös varastoon tehty työ on hukkaa.
- Ihmiskeskeisyys (center on the people who add value): keskity ihmisiin, jotka tuottavat lisäarvon, ei siis niinkään johtoportaaseen, vaan ruohojuuritason tekijöihin. Tähän liittyy arvostus, tehokas kommunikointi, hyvä työilmapiiri sekä jatkuva koulutus ja toiminnan tehostaminen.

Leanin ajattelutapa on keskeistä ketterissä menetelmissä. Hukkaa on etukäteen tuotettu määrittelydokumentti, jos ominaisuutta ei toteuteta ja varastoon tehty työ, jos toteutus tapahtuu vasta kaukana tulevaisuudessa. (Haikala & Mikkonen 2011, 54–55)

Kanbanin kolme periaatetta:

- Työn kulun visualisointi: kukin työ on omalla kortillaan ja kortti siirtyy sarakkeesta toiseen sen mukaisesti, missä vaiheessa toteutus on (tehtävätaulu).
- Kaistanrajoittimet: kussakin vaiheessa samanaikaisesti olevien töiden määrä on rajoitettu.
- Läpimenoajan mittaaminen: kehitysprosessia voi optimoida, jos mitataan työn kussakin vaiheessa viettämä aika.

Kanbanissa ei ole pyrähdyksiä, joten uusi tehtävä voidaan aloittaa aina, kun on tilaa. Yksi pyrähdysten eduista on tiimin rauhoittaminen vaatimusmuutoksilta pyrähdyksen ajaksi. Tästä voi kuitenkin olla haittaa, jos palvelu on rajoitetussa käytössä koko kehitystyön ajan ja käyttäjien uudet ideat tulisi saada heti koekäyttöön (perpetual beta). Scrumin ja Kanbanin yhdistelystä käytetään nimeä Scrumban. (Haikala & Mikkonen 2011, 55–56)

2 OHJELMISTOTESTAUS

Ohjelmistotestaus on osa ohjelmistotuotantoa, mutta sillä on omat periaatteensa sekä osa-alueensa. Tässä luvussa käydään tarkemmin läpi luvussa 1 esitelty ohjelmistotuotannon osa-alue: testaus.

2.1 Ohjelmistotestauksen periaatteet

Testauksen perusperiaatteet on kiteytetty ISTQB-testaussertifikaatissa (International Software Testing Qualifications Board, 2013), jossa ohjelmistotestaukselle on asetettu 7 periaatetta.

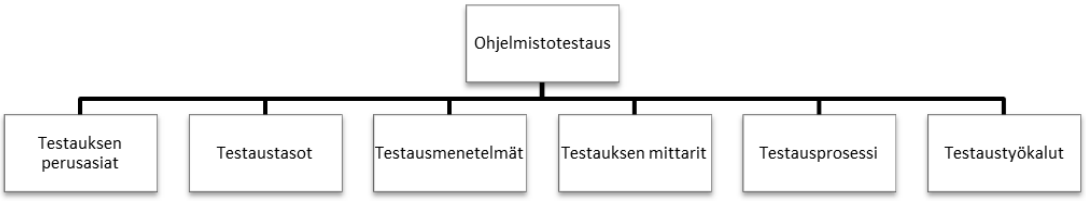
Taulukko 2. Testauksen periaatteet (Kasurinen 2013, 48–49)

	Periaate
1	Testaus osoittaa vikojen olemassaolon ja pienentää todennäköisyyttä sille, että ohjelmasta löytyy edelleen vikoja. Se, että testaus ei löydä yhtään vikaa ei takaa tuotteen viattomuutta.
2	Täydellinen testaus on mahdotonta. Testauksen tulee perustua riskien kartoittamiseen sekä tehtävän testaustyön priorisointiin.
3	Aikainen testaus. Testaus pitää aloittaa jo esituotantovaiheessa ja sen pitää kattaa myös vaatimusmäärittelyt ja projektia varten tehdyt suunnitelmat.
4	Vikojen kasaantuminen. Testauksen painopisteet tulisi sijoittaa niihin osiin, joissa tiedetään tai odotetaan olevan eniten vikoja.
5	Hyönteismyrkkyparadoksi. Testitapauksia tulee päivittää, lisätä ja kehittää projektin edetessä, muuten ne menettävät vaikutuksensa.
6	Testaus on tilanneriippuvaista. Käytetyn prosessin tai hallintamallin pitää sisältää liikkumavaraa, jotta erilaiset tilanteet saadaan selvitettyä ilman mallin muutoksia.
7	Virheettömyyden harhaluulo. Testaus, laadunvalvonta tai virheiden korjaaminen ei auta, jos tuote on suunniteltu väärin tai ei täytä kaikkia sille asetettuja odotuksia.

Taulukon 2 perusperiaatteet luovat puitteet, joiden päälle ohjelmistotestaus rakentuu. Ne määrittävät ja perustelevat ohjelmistotestauksessa käytettyjä toimintatapoja.

2.2 Ohjelmistotestauksen osa-alueet

Ohjelmistotestaus koostuu osa-alueista, joita voidaan mallintaa esimerkiksi IEEE (Institute of Electrical and Electronics Engineers) SWEBOK (Software Engineering Body of Knowledge) – mallin avulla.



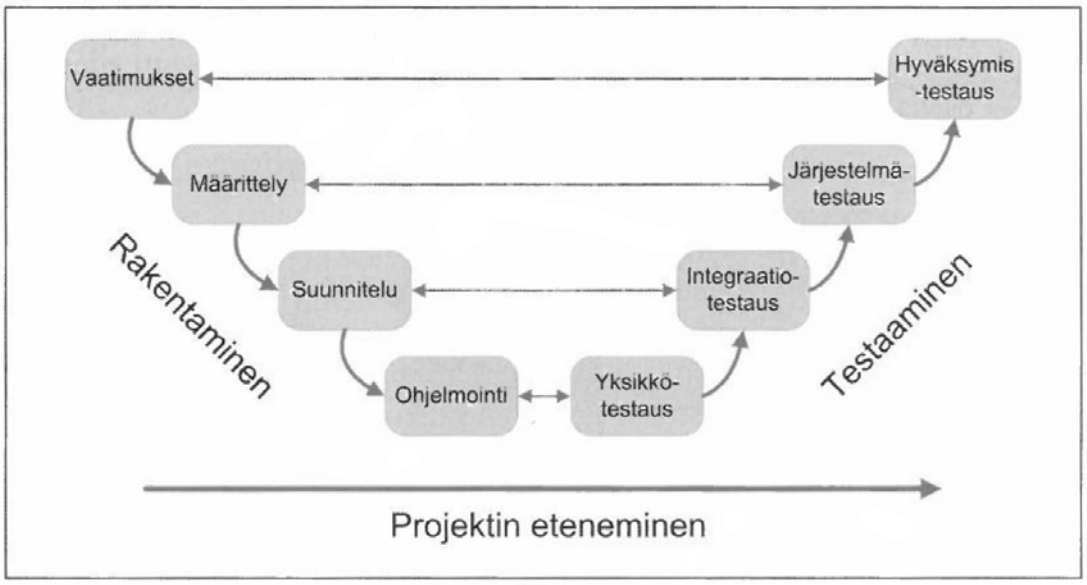
Kuva 3. SWEBOK-mallin testaamisen osaamisalueet (Kasurinen 2013, 46)

Kuvan 3 osaamisalueet kattavat kaikki toiminnot, jotka ovat keskeisiä ohjelmistotestauksen tekemiselle.

2.2.1 Testauksen perusasiat

Testauksen perusteita ovat käytetty terminologia, avainasiat, kuten testattavuus, testauksen rajoitteet, testauksen tehokkuuden arvioiminen, testitapausten valintamenetelmät sekä suhde muuhun ohjelmistokehitykseen ja laadunhallintaan. (Kasurinen 2013, 46–47)

2.2.2 Testaustasot



Kuva 4. V-malli (Kasurinen 2013, 51)

Kuva 4. esittää testauksen V-mallia, joka määrittelee testauksen eri vaiheet: yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksymistestaus. Näitä vaiheita kutsutaan testaustasoiksi, koska niissä testausta tehdään eri tasoilla. (Kasurinen 2013, 51)

2.2.3 Testausmenetelmät

Testaustyön tekemiseen on olemassa monia eri työtapoja eli testausmenetelmiä. Menetelmät voidaan jakaa sen mukaan missä projektin vaiheessa niitä käytetään. Projektin suunnitteluvaiheen testausmenetelmä on esimerkiksi prototyyppi testaus, jolla voidaan testata käyttöliittymää ja keskeisiä käyttötapauksia.

Kehitysvaiheen menetelmiä ovat mm. musta-, harmaa- ja lasilaatikko testaus, sekä regressiotestaus. Kehitysvaiheen loppupuolella, lähes valmiille tuotteelle, on omat testausmenetelmänsä. Näitä ovat mm. käytettävyys-, kuormitus-, suorituskyky-, sekä betatestaus. (Kasurinen 2013, 60–75)

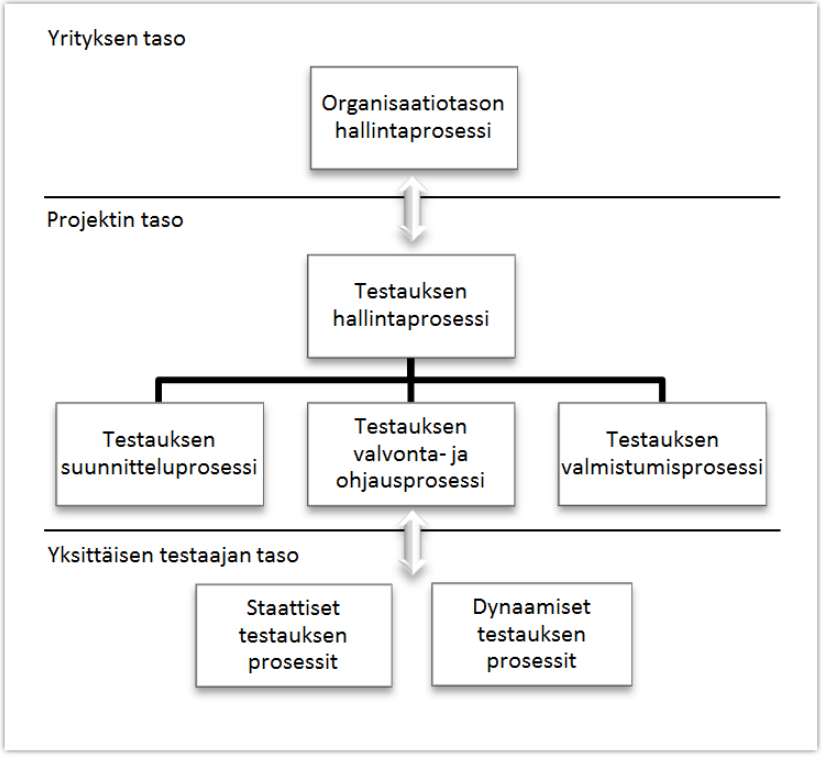
2.2.4 Testauksen mittarit

Testauksen etenemistä ja kattavuutta arvioidaan ja seurataan mittareilla. ISO/IEC 29119 (International Organization for Standardization / International Electro technical Commission) jakaa mittarit 7 eri luokkaan käytetyn tietolähteen perusteella: raaka testausdata, työnsuunnittelu, -kattavuus, projektin tila, testitulokset, testauksen laatu ja luotettavuus. Esimerkiksi budjettiseuranta ja työajanseuranta ovat raakadataan perustuvia mittareja.

Projektissa käytettävän mittarin tulisi hyödyntää tietoa, joka muodostuu joka tapauksessa tehdystä työstä, ilman että se aiheuttaa testaajalle erityistä työtä. (Kasurinen 2013, 162)

2.2.5 Testausprosessi

Testausprosessi vaihtelee käytetyn toimintamallin mukaan, mutta ISO/IEC 29119 standardimalli on hyvä esimerkki testauksen keskeisistä toiminnoista.



Kuva 5. Testausprosessit (Kasurinen 2013, 106)

ISO/IEC 29119 standardimalli (Kuva 5) jakaa testausprosessit ja niistä syntyvän dokumentaation organisaatio- ja projektitasoille. Kummallakin tasolla on hallintaprosessi, projektitasolla sen alle kuuluu testauksen suunnittelu-, valvonta- ja ohjaus-, sekä valmistumisprosessi.

Yksittäisen testaajan taso voidaan vielä jakaa staattisiin (valmistelu, dokumentointi, raportointi) ja dynaamisiin (käytännön testaustyö) testausprosesseihin. (Kasurinen 2013, 106–107)

2.2.6 Testaustyökalut

Testauksessa käytetään enimmäkseen samoja työvälineitä kuin muussakin ohjelmistokehityksessä. Testaajan tärkeimmät työkalut ovat kehitysympäristö, usein sama, millä itse ohjelman kehitys on tehty, sekä kommunikointivälineet, joilla suoritetaan lisätiedustelut sekä ilmoitetaan havaituista ongelmista eteenpäin.

Vikatietokanta on tärkeä testauksen hallinnan työväline. Kantaan raportoidaan löydetyt virheet (bugit) ja sen avulla seurataan testauksen ja korjaustyön etenemistä, sekä arvioidaan testaustyön tehokkuutta ja laatua. Usein sen avulla hallitaan myös työtehtäviä.

Muita työvälineitä ovat mm. tyngät (test stubs, dummys) eli yksinkertaistetut sijaiskomponentit, joita hyödynnetään järjestelmän testaamisessa oikeiden komponenttien sijasta. Analysaattorit taas ovat testauksen apuvälineitä, jotka tarkastavat järjestelmän toimivuutta tunnettujen ongelmien varalta. Tyypillisiä tarkastelun kohteita ovat ohjelmointikäytännöt, syntaksi ja testitapausten koodikattavuus. (Kasurinen 2013, 84–87)

3 AUTOMAATIO

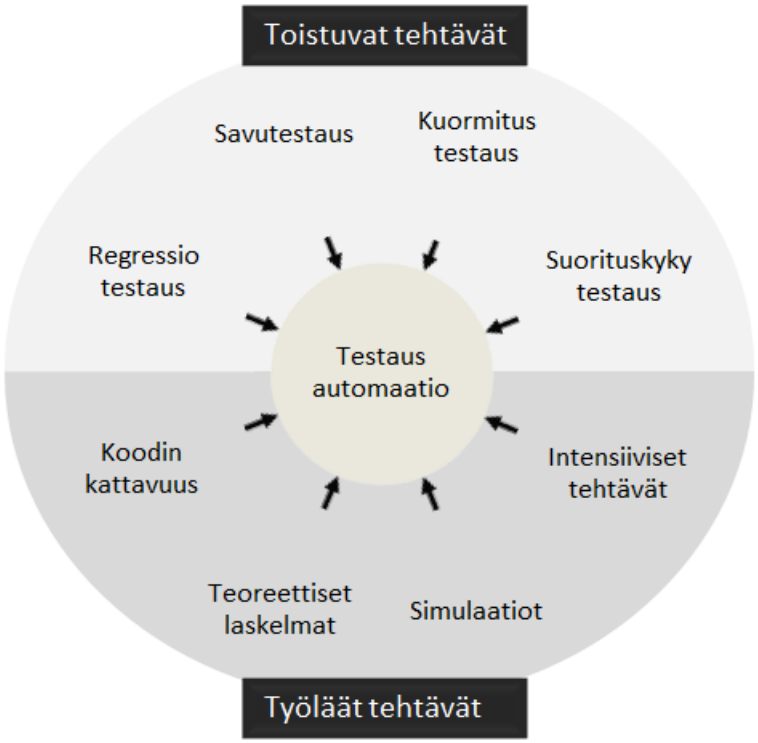
Käsite automaatio liitetään yleensä teollisuusautomaatioon ja sen tuotantoprosesseissa käytettyihin koneisiin sekä robotiikkaan. Tässä luvussa käsitellään kuitenkin automaatiota ohjelmistotuotannon kannalta, kuten automatisoitua ohjelmistotestausta, ohjelmiston dokumentoinnin automaattista generointia, sekä jatkuvaa integrointia.

3.1 Testausautomaatio

Testausautomaatio on testaustoiminnan muoto, jossa ohjelman testaamista varten rakennetaan automaatiotyövälineitä testien tekemistä varten. Testauksen automatisoinnin tavoitteena on ottaa joukko toistuvasti tehtäviä tai ihmiselle työläitä testitapauksia ja rakentaa niiden nopeaa tarkastamista varten erillinen testausympäristö.

Pohjimmaisena ajatuksena automatisoinnissa on vapauttaa testaajat muihin tehtäviin. Jos tuotteesta tehdään esimerkiksi joka yö uusi käännös (daily build), ei testaajien ajankäytön kannalta ole järkevää käyttää joka päivä aikaa samojen perustestien tekemiseen. (Kasurinen 2013, 76)

Testausautomaation työkalut ovat usein käytännössä täysin itse rakennettuja. Yleensä testausautomaatio rakentuu joukosta sarja-ajettavia testiajureita, jotka lähettävät testattavalle komponentille viestejä ja kirjaavat ylös mitä tapahtui, sekä vastasiko reaktio annettuja ehtoja. Avoimen lähdekoodin ratkaisu on esimerkiksi testausautomaatio Jenkins. (Kasurinen 2013, 86)



Kuva 6. Mitä kannattaa automatisoida (Craig & Jaskiel 2002, 218)

Kaikkea ei kannata automatisoida, sillä automaattisen testin tekeminen vaatii usein enemmän osaamista ja aikaa, kuin manuaalisen testin tekeminen. Automaattisen testin ajoon kuluvan ajan ja ajokertojen suhteen vertailun avulla voidaan määrittää, kannattaako testitapaus automatisoida vai ei. Tätä nyrkkisääntöä voidaan käyttää valittaessa automatisoitavia testitapauksia. Esimerkiksi vain kerran ajettavat testit kannattaa lähes aina ajaa manuaalisesti. (Craig & Jaskiel 2002, 217)

3.2 Generatiivinen dokumentointi

Generatiivisella tai automaattisella dokumentoinnilla tarkoitetaan ohjelmiston dokumenttimuotoisen kuvauksen automaattista tuottamista ohjelmakoodista. Myös erilaisia dokumenttipohjia voidaan generoida automaattisesti.

Automaattinen rajapintadokumentaation generointi ohjelmakoodista on mahdollista käyttäen sopivia työkaluja, kuten Doxygen tai Javadoc, olettaen kuitenkin, että kehittäjät käyttävät työkalun määrittämiä määrämuotoisia kommentteja.

Opinnäytetyön tukityökalua voidaan käyttää automaattisen rajapintadokumentaation generointiin. Toiminnaltaan se eroaa hieman edellä mainituista työkaluista, sillä se ei käytä määrämuotoisia kommentteja, vaan tukeutuu enemmänkin käytettävän koodikielen olemassa olevaan syntaksiin. Lisäksi työkalu hyödyntää organisaation ohjelmointikäytäntöjä.

Dokumentoinnin tuottamien automaattisesti on hyödyllistä, sillä dokumentit tulisi saada ajan tasalle mahdollisimman pian muutoksen tullessa, muuten niistä ei ole juurikaan hyötyä jatkossa. Dokumentointityön jäädessä projektin loppuvaiheeseen, erityisesti manuaalinen kuvausten päivittäminen jää helposti hoitamatta projektin päättymiseen liittyvien kiireiden pakottaessa priorisoimaan toimintoja. (Haikala & Mikkonen 2011, 195)

Testausvaiheessa syntyy paljon dokumentoitavia asioita ja yksi testauksessa käytetty työkalu onkin dokumenttipohjat, joita käytetään testitapausten määrittelyyn, raportointiin sekä ohjaukseen. Kuten rajapintadokumentti, dokumenttipohjakin voidaan generoida automaattisesti, mutta se voi olla myös pelkkä täytettävä lomake tai word-dokumentin runko. Käytettäessä automaattista testausta dokumentointi, erityisesti raportointi, on yleensä automatisoitua.

Dokumenttipohjien tarkoituksena on varmistaa, että kaikki millään tavalla arvokas tieto tulee kirjattua ylös ja että testaajien aika ei mene dokumenttien muotoilemiseen vaan tehokkaasti joko testaamiseen tai vikojen raportointiin. (Kasurinen 2013, 88)

3.3 Jatkuva integrointi

Jatkuva integrointi (continuous integration) tarkoittaa sitä, että kehittäjät toimivat saman ohjelmistoversion kanssa ja kaikki muutokset talletetaan versionhallintaan mahdollisimman nopeasti. Tämä pakottaa muutokset niin pieniksi, että ne lähes aina toimivat, jolloin versionhallinnassa oleva kokonaisuus on aina valmis rakennettavaksi ja suoritettavaksi. Toisin sanoen muutokset ovat niin pieniä ja nopeita, että konflikteja ei ehdi syntyä tai ne ovat helposti ratkaistavissa.

Jotta järjestelmän toimivuus voidaan varmistaa, liittyy jatkuvaan integrointiin käytännössä aina automatisoitu testaus. Kun testitapauksia tai ohjelmaa muutetaan, järjestelmä rakentaa suoritettavan version ohjelmasta ja ajaa testit automaattisesti.

Kehittäjän kannalta jatkuva integrointi muuttaa asennoitumista testien suorittamiseen. Kyse ei ole enää laadunvarmistuksesta korkealla abstraktitasolla, vaan pientenkin asioiden huolellisesta testaamisesta. Ongelmat saadaan yleensä selvitettyä nopeammin, sillä välittömästi muutoksen jälkeen muutettu kohta ja muutoksen tarkoitus on paremmin mielessä kuin myöhemmin jotain suurempaa kokonaisuutta testattaessa.

Jatkuvaa integrointia suunnitellessa tulee huomioida, että jos käännös tai testit eivät mene läpi, koko järjestelmä on perustavanlaatuisesti rikki. Koska kukaan kehittäjä ei pysty lisäämään uusia ominaisuuksia ohjelmaan ennen kuin ongelmat on korjattu, vaatii tämä välitöntä huomiota kehittäjältä, jonka muutokset ongelman aiheuttivat, mutta usein myös koko kehitysryhmältä.

Toinen ongelma liittyy testien suoritusaikaan. Koska pienikin muutos aiheuttaa järjestelmän uudelleen rakentamisen ja testien uudelleensuorituksen, voidaan testejä joutua optimoimaan, tai ajamaan rinnakkain suoritusajan lyhentämiseksi. Jatkuvassa integroinnissa usein käytettyjä työkaluja ovat muun muassa Cruise Control ja Hudson. (Haikala & Mikkonen 2011, 175–176)

4 TYÖKALUN SPESIFIKAATIOT

4.1 Yleiskuvaus

Tukityökalu luo ohjelmistokoodista selainpohjaisen dokumentin, joka sisältää koodiin linkitetyn listauksen moduuleista löytyneistä koodin elementeistä. Mikäli elementtejä on koodissa kommentoitu, myös kommentit otetaan mukaan listaukseen. Työkalu huomauttaa puuttuvista kommenteista. Työkalun luomien listojen järjestämistä ja kopioimista varten on olemassa omat funktiot.

4.1.1 Käytetyt ohjelmat

Toteutus ja testaus suoritettiin seuraavilla ohjelmilla: Microsoft Visual Studio, Google Chrome, Mozilla Firefox ja Internet Explorer.

4.1.2 Käytetyt kielet

Tukityökalu toteutuksessa C++ kieltä käytettiin Windows-lomakkeiden ja datan keräämiseen. Selainpohjainen käyttöliittymä taas tehtiin JavaScriptin (jQuery), HTML5:n sekä CSS:n avulla.

4.1.3 Käyttö

Tukityökalun käyttäjiä ovat projektin ohjelmisto-, sekä testausinsinöörit. Työkalua voidaan käyttää itsenäisesti, tai sen luoma dokumentti voidaan ladata palvelimelle yhteiseen jakoon. Työkalu ei vaadi käyttäjältä erityisosaamista. Työkalua voidaan käyttää päivittäin, tai aina uuden ohjelmistoversion luonnin yhteydessä.

4.1.4 Oletukset ja riippuvuudet

Tukityökalu toimii Windows käyttöjärjestelmässä ja vaatii toimiakseen selaimen. Työkalu käyttää oletuksena käyttäjän asettamaa oletusselainta.

4.2 Arkkitehtuuri

Ohjelman koodin arkkitehtuuri on jaettu kolmeen eri tiedostoon, “Source.cpp”, “Library.cpp” ja “Header.h”. Source.cpp sisältää pääfunktion ”WinMain” joka sisältää ”pääloopin” jossa haetaan kerätyistä tiedostoista halutut avainsanat ja kommentit. nämä kerätään ja niiden avulla muodostetaan index.html tiedosto. WinMainin sisällä ajetaan kaikki muut funktiot.

Taulukko 3. Tärkeimmät funktiot

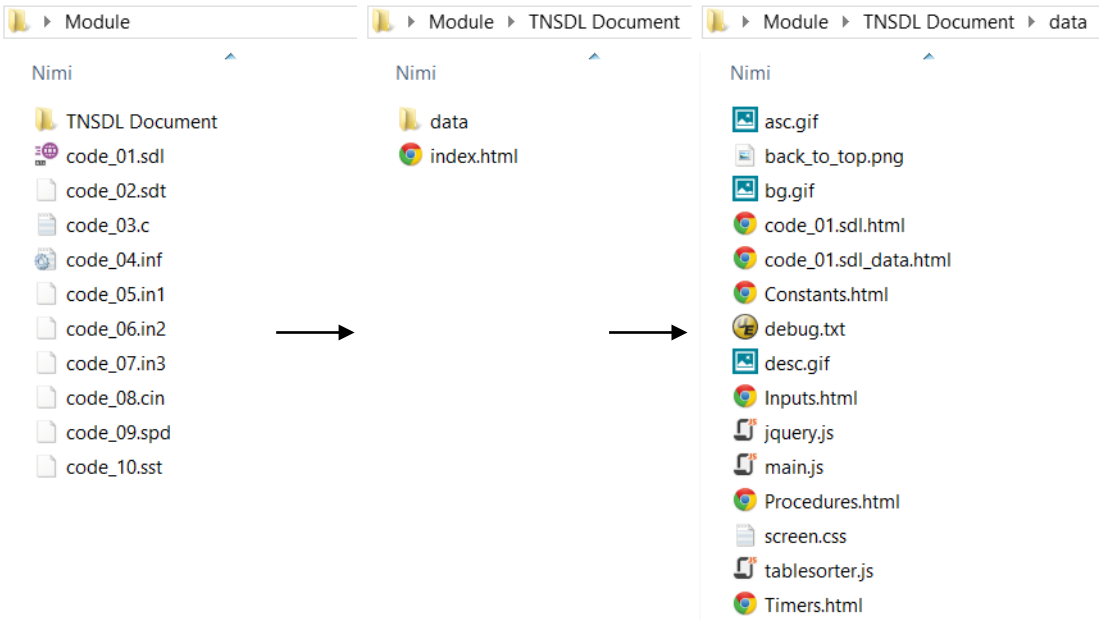
Funktio	Toiminta
get_directory()	Luo Windows -lomakkeen, jonka avulla käyttäjä valitsee koodit sisältävän kansion. Käyttäjän valitsema kansio palautetaan merkkijonona pääohjelmalle.
get_files()	Hakee kooditiedostot annetusta kansiosta.
progress_bar()	Luo ja päivittää käyttäjälle näkyvää latauspalkkia.
create_stylesheet()	Luo selaimen käyttämän CSS – merkintäkieli tiedoston, jonne on määritelty käyttöliittymän ulkoasu.
create_jquery()	Luo selaimen käyttämän jQuery – skripti tiedoston, jonne on määritelty käyttöliittymän tarvitsemat skriptit.
create_tablesorter()	Luo käyttöliittymän taulukoiden käyttämän tablesorter skripti tiedoston joka mahdollistaa taulukoiden järjestelyn.
create_main()	Luo selaimen pääsivun index.html.
create_list()	Luo listat kaikista löydetyistä proseduureista, inputeista, ajastimista ja vakioista.
create_page()	Luo jokaiselle löydetylle kooditiedostolle oman sivun, josta löytyy kaikki koodista löydetyt avainsanat.
create_iframe_file()	Upottaa kooditiedoston koodin selaimelle ja linkittää siihen kaikki löydetyt avainsanat.

4.3 Toiminnot

Työkalu hakee annetusta moduulista avainsanoja ja tekee niistä koodiin linkitetyn listauksen. Tässä tapauksessa avainsanoja ovat: procedures, inputs, timers ja constants, mutta käyttötarpeesta riippuen eri avainsanoja voidaan määrittää. Tästä lisää jatkokehitys kappaleessa.

4.3.1 Tiedostorakenne

Työkalu luo käyttäjän osoittamaan, koodit sisältävään, kansioon ”TNSDL Document” hakemiston, jonka alle se sijoittaa dokumentin ”index.html”, sekä ”data” kansion, joka sisältää html-, css-, javascript-, sekä kuvatiedostot.



Kuva 7. Tiedostorakenne

4.3.2 Listaus

Työkalu luo listan jokaisesta löydetystä avainsanasta. Listaan tulee avainsanasta riippuen: nimi, kommentti, arvo, sanoman numero, vakio ja muuttuja(t). Listat tehdään tiedosto- ja moduulikohtaisesti.

4.3.3 Linkitys

Virheenetsinnän ja testauksen helpottamiseksi jokainen listan elementti on linkitetty kyseiseen kohtaan koodissa.

4.3.4 Kopiointi

Listan muokkaaminen kopiointia varten on automatisoitu, jotta siirto tekstinkäsittely-ohjelmaan onnistuu ilman käyttäjän väliintuloa.

4.3.5 Muut ominaisuudet

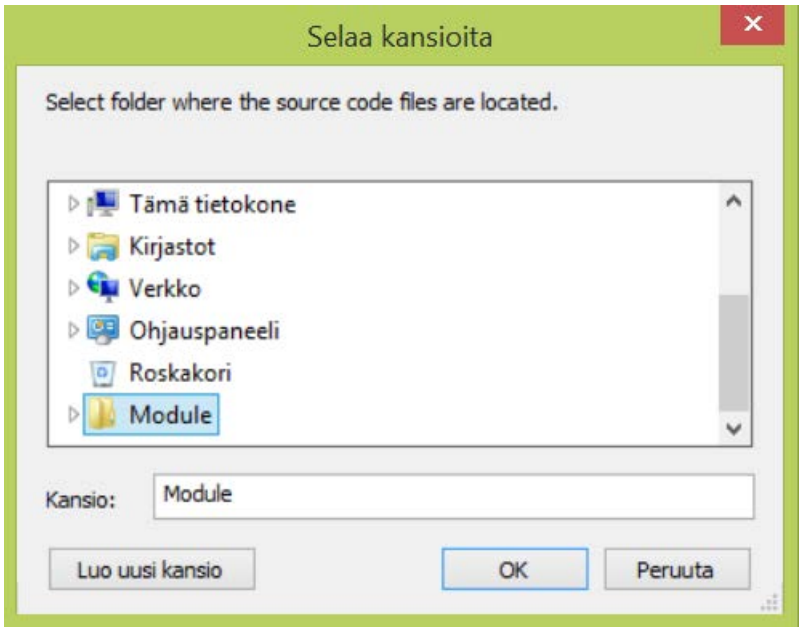
Työkalu hyödyntää kaikkia selaimen olemassa olevia toimintoja, esimerkiksi hakutoimintoa.

4.4 Käyttöliittymä

Käyttöliittymän tulee olla yksinkertainen ja mahdollisimman monipuolinen. Tästä syystä tukityökalun käyttöliittymänä toimivat olemassa olevat Windows-lomakkeet, sekä käyttäjän valitsema oletusselain.

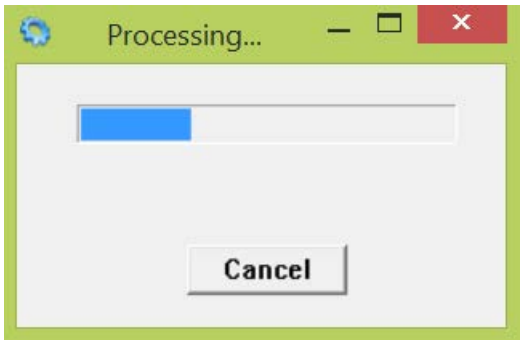
4.4.1 Windows-lomakkeet

Käyttäjän käynnistäessä työkalun, aukeaa lomake, jonka avulla valitaan koodit sisältävä kansio.



Kuva 8. Kansion valinta

Tämän jälkeen työkalu alkaa käsittelemään dataa ja latausikkuna aukeaa.

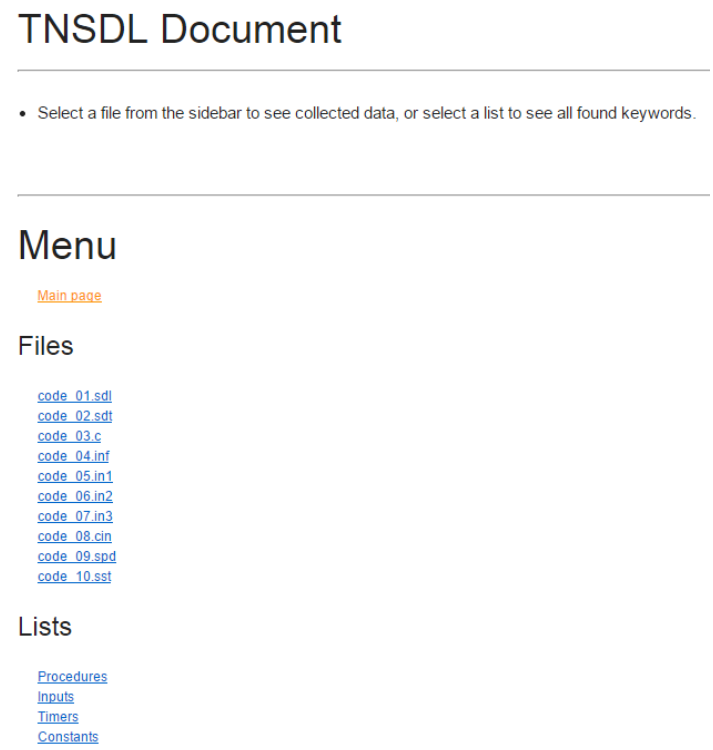


Kuva 9. Latausikkuna

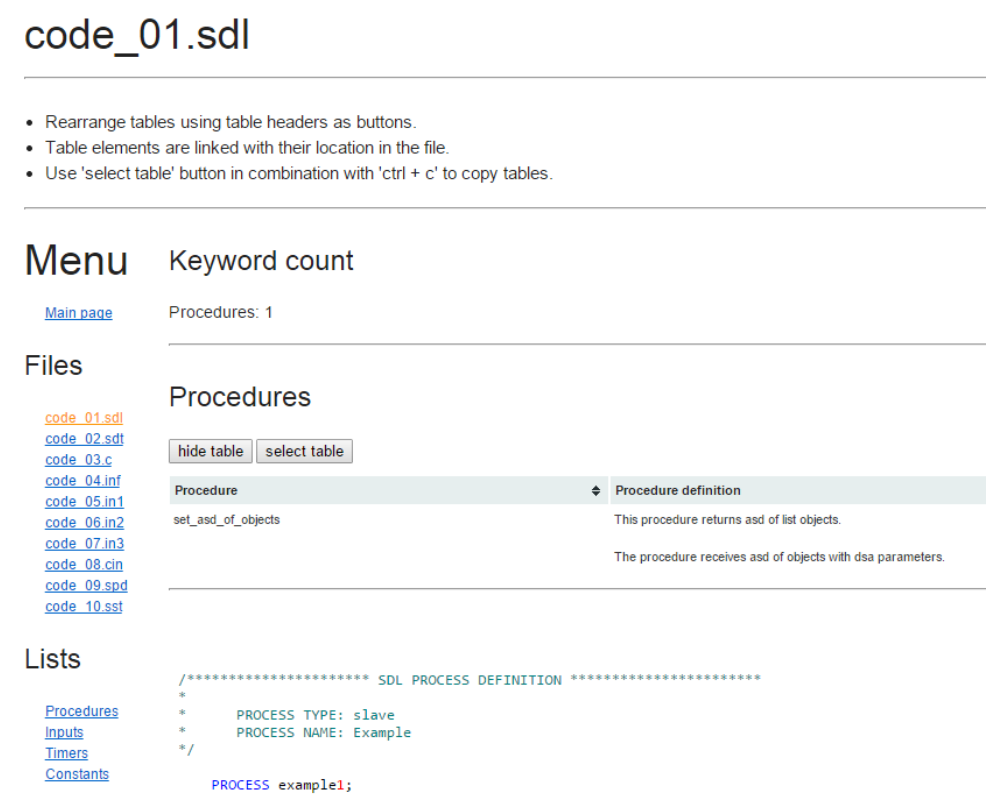
Latauksen valmistuttua, työkalu avaa dokumentin oletusselaimella.

4.4.2 Selainnäkymä

Jokaisella sivun yläreunasta löytyy otsikko ja pieni ohje-osia. Valikko sijaitsee näytön vasemmassa reunassa ja muodostuu etusivusta, tiedostoille luoduista sivuista sekä listoista. Sivun keskelle tulevat näkyviin löydettyjen avainsanojen lukumäärät, listat ja koodi. Oikeasta alareunassa sijaitsee ”back to top” painike, jolla pääsee takaisin dokumentin yläosaan.



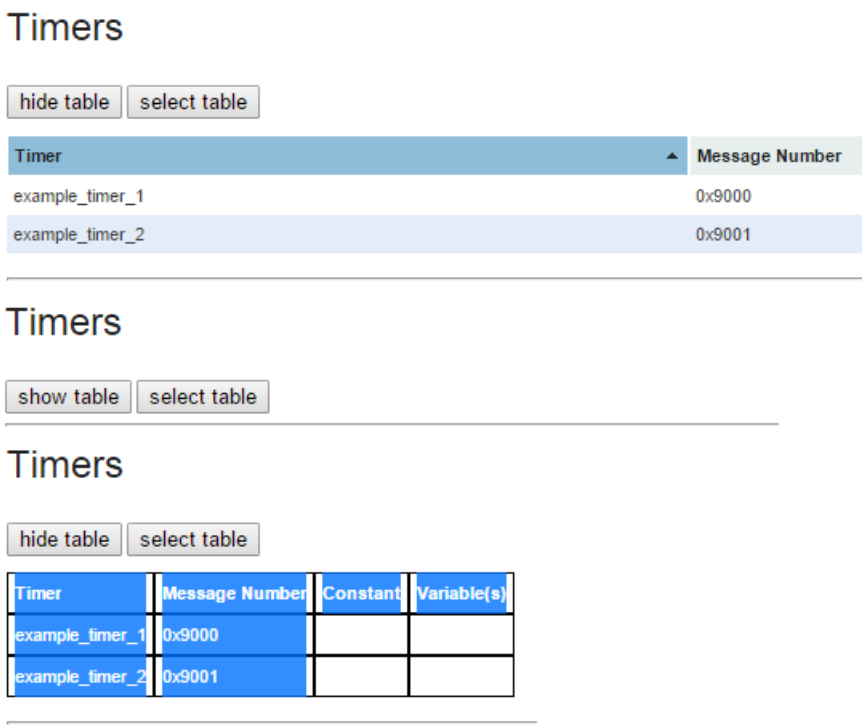
Kuva 10. Valikkorakenne



Kuva 11. Sivunäkymä

4.4.3 Lista

Oletuksena avainsana lista järjestää elementit samassa järjestyksessä, kuin missä ne ilmestyvät koodissa. Lista voidaan järjestää sekä aakkos-, että suuruusjärjestykseen. Listan yläpuolelta löytyy painikkeet, joilla listan voi avata/sulkea tai valita listan kopiointia varten. Valittaessa listaa kopioitavaksi, se pitää sen hetkisen järjestyksensä.



Kuva 12. Lista

4.4.4 Koodi

Klikattaessa listan elementtiä, siirtyy käyttäjän näkymä siihen kohtaan koodia, mistä elementti löytyi. Koodista on väreillä eroteltu avainsanat, numerot, kommentit ja merkkijonot.

```
ALTERNATIVE mod_test_log;
(2):
  TASK asd_r( log_type_t_test_c,
                    @'set_asd_of_objects:'
                     'asd1 : %d'
                     'asd2 : %d'
                     ,asd4, asd5);
ENDALTERNATIVE;
```

Kuva 13. Koodi

4.5 Rajoitukset

Työkalu toimii Windows ympäristössä, versioilla Windows XP – Windows 8.1. Työkalu tukee kolmea isointa selainta:

Taulukko 4. Tuetut selaimet

Selain	Versio
Google Chrome	24.0.1312 →
Mozilla Firefox	18.0 →
Internet Explorer	11.0 →

4.6 Hylätyt ratkaisut

Yhtenä ajatuksena oli html-dokumentin sijaan tuottaa kerätystä datasta valmis Word-dokumentti, mutta näin olisi jätetty hyödyntämättä työkalun mahdollisia ominaisuuksia ja rajattu työkalu osaksi Office-pakettia.

4.7 Jatkokehitys

Työkalulla on paljon tilaa kehittyä. Tärkeimpänä kehityksenä olisi muokata työkalu toimimaan muille ohjelmointikielille.

Ohjelmistokehityksen kannalta mielenkiintoista voisi olla esimerkiksi, vektorigrafiikalla luotavat sekvenssikaaviot, joilla voitaisiin seurata koodin kulkua proseduurista toiseen. Myös proseduurien muistinkäyttöä voitaisiin seurata.

Kerättävien avainsanojen määrää voitaisiin kasvattaa. Koodin kommentteja voitaisiin kerätä tehokkaammin ja laajemmin, ottamaan huomioon esimerkiksi korjattuja koodikohtia.

Käyttöliittymää voitaisiin parannella, mm. korostamalla koodikohta mihin listalta siirryttiin.

5 YHTEENVETO

Sain tukityökalun toteutettua aikataulussa ja opin paljon sitä tehdessäni. Työ oli haasteellinen, mutta palkitseva, erityisesti silloin kun pitkään mietityttäneen ongelman sai ratkaistua.

Tukityökalusta kehittyi lopulta toimiva osa ohjelmistokehitystä ja näkisin, että jatkamalla työkalun kehitystä siitä saataisiin toimiva ratkaisu moneen tulevaisuuden haasteeseen ohjelmistokehityksen saralla.

LÄHTEET

Haikala, I. & Märijärvi, J. 2006. Ohjelmistotuotanto. Helsinki: Talentum.

Haikala, I. & Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Helsinki: Talentum.

Kasurinen, J. P. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.

Craig, R & Jaskiel, S. 2002. Systematic Software Testing. Boston: Artech House.