POLARION ALM VALTRAN OHJELMISTOKEHITYKSEN TUKENA

Tero Pesonen

Opinnäytetyö
Huhtikuu 2012

Ohjelmistotekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) PESONEN, Tero	Julkaisun laji Opinnäytetyö	Päivämäärä 10.04.2012
	Sivumäärä 35	Julkaisun kieli Suomi
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)

Työn nimi
POLARION ALM VALTRAN OHJELMISTOKEHITYKSEN TUKENA

Koulutusohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)
SALMIKANGAS, Esa

Toimeksiantaja(t)
Valtra Oy Ab

Tiivistelmä

Opinnäytetyön tarkoitus oli käyttöönottaa ja mukauttaa ohjelmiston elinkaarenhallinnan työkalu Polarion ALM opinnäytetyön toimeksiantajalle Valtra Oy Ab:lle. Polarionin käyttöönoton tavoitteena oli tukea Valtran ohjelmistokehitystä parantamalla ohjelmiston elinkaaren eri vaiheiden ja tuotosten jäljitettävyyttä toisiinsa sekä kykyä suunnitella ja seurata edistymistä.

Opinnäytetyön teoriaosassa selvitettiin lyhyesti ohjelmiston elinkaarenhallinnan taustaa. Valtran ohjelmistokehityksen tilannetta tarkasteltiin ennen Polarionin käyttöönottoa ohjelmiston elinkaarenhallinnan näkökulmasta. Ohjelmistokehitykseen liittyy paljon vaiheita ja käytettäviä työkaluja, joten opinnäytetyössä keskityttiin tarkastelemaan Valtran erityispiirteitä vaatimustenhallintaan, ohjelmiston testaukseen, vikojen seurantaan ja ohjelmiston julkaisuihin liittyen. Näillä osa-alueilla tunnistettiin haasteita ohjelmiston elinkaarenhallinnan kannalta tärkeiden jäljitettävyyden, raportoinnin ja työkalujen integraatioiden osalta.

Opinnäytetyön käytännön toteutuksen, eli Polarionin käyttöönoton ja mukautuksen, lähtökohtana olivat ohjelmiston elinkaarenhallinnan teoria ja Valtran ohjelmistokehityksen tilanne ennen käyttöönottoa. Keskeisessä osassa toteutusta olivat Polarionin asetusten määrittely, jäljitettävyyttä ja edistymisen seurantaa tukevat näkymät ja raportit sekä työkalujen integraatiot.

Polarion otettiin käyttöön ja mukautettiin tukemaan Valtran erityispiirteitä opinnäytetyön aikana, joten opinnäytetyön pääasiallinen tavoite saavutettiin. Polarionin käytöllä saavutettuja hyötyjä havaittiin jo opinnäytetyön aikana mm. ohjelmistokehitykseen liittyvien kehitys- ja muutospyyntöjen käsittelyssä. Valtralla kehitettävän ohjelmiston elinkaari on kuitenkin pitkä, joten ohjelmiston elinkaarenhallinnan työkalun käytön hyödyt jakautuvat myös pitkälle ajalle ja ovat näin ollen vaikeita arvioida heti käyttöönoton jälkeen. Polarionin kaikkia ominaisuuksia ei hyödynnetty vielä opinnäytetyön aikana, joten jatkokehitykselle jäi vielä mahdollisuuksia.

Avainsanat (asiasanat)

Ohjelmiston elinkaarenhallinta, vaatimustenhallinta, jäljitettävyys, Polarion ALM

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
PESONEN, Tero	Bachelor´s Thesis	10.04.2012
	Pages 35	Language Finnish
	Confidential ( ) Until	Permission for web publication (X)

Title
POLARION ALM SUPPORTING VALTRA'S SOFTWARE DEVELOPMENT

Degree Programme
Bachelor's Degree in Software Engineering

Tutor(s)
SALMIKANGAS, Esa

Assigned by
Valtra Inc

Abstract

The purpose of the Bachelor's Thesis was to deploy and adjust application lifecycle management tool Polarion ALM to support the software development in Valtra Oy Ab. The objective of the Polarion deployment was to improve software development by providing traceability in lifecycle activities and ability to plan and monitor progress of development.

The theory and methods of application lifecycle management were discussed briefly in the theory part of the thesis. Valtra's software development before the Polarion deployment was examined from the perspective of application lifecycle management. The examination focused mostly on requirements management, testing, issue tracking and release management. Challenges were found in important application lifecycle management disciplines, such as traceability, reporting and tool integrations.

The deployment and adjustments of Polarion were based on the theory of application lifecycle management and examination of Valtra's software development methods before the deployment. Essential parts of the implementation were definition of Polarion settings, tool integrations and development of views for traceability, planning and monitoring of progress.

Successful deployment and adjustment of Polarion was the main goal of the Bachelor's Thesis and it was accomplished. Some of the benefits of the tools for application lifecycle management tools span over the entire lifecycle of software, thus it became hard to evaluate the benefits right after the deployment. Not all features of Polarion were utilized during the Polarion deployment, therefore there are still opportunities to expand the usage of Polarion in the future.

Keywords

Application Lifecycle Management, Requirements Management, Traceability, Polarion ALM

Miscellaneous

SISÄLTÖ

KUVIOT

TAULUKOT

1 OPINNÄYTETYÖN LÄHTÖKOHDAT

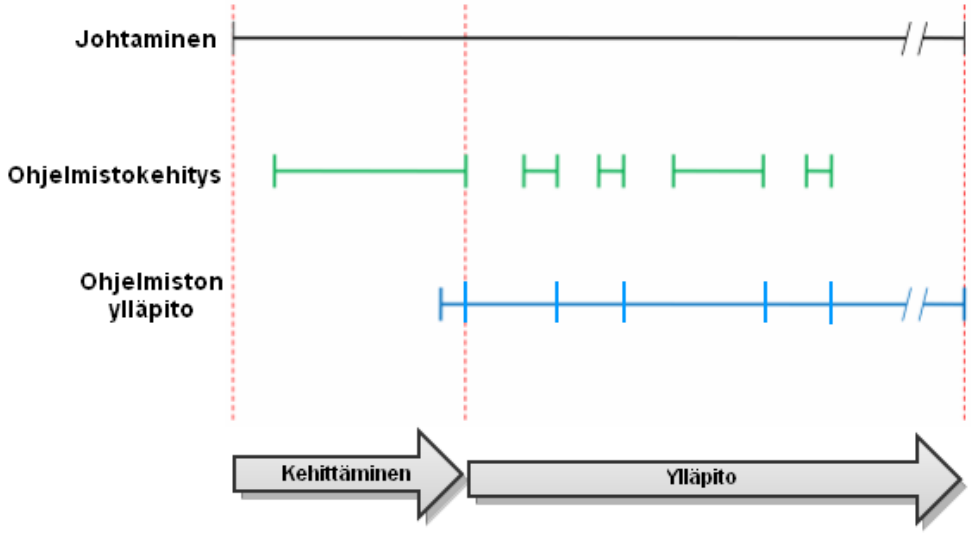
Opinnäytetyön toimeksiantajana toiminut Valtra Oy Ab kehittää, valmistaa, markkinoi ja huoltaa Valtra-traktoreita. Valtra toimii osana kansainvälistä AGCO-yhtymää, joka on yksi maailman suurimmista maatalouskoneita ja niiden varaosia suunnittelevista, valmistavista ja myyvistä yrityksistä. AGCO tarjoaa erilaisia maatalouskoneita useilta eri tuotemerkeiltä ja sen tuotteita myy n. 3900 jälleenmyyjää yli 140 maassa. (Tietoa Valtrasta n.d.)

Opinnäytetyön tarkoitus oli käyttöönottaa ja mukauttaa ohjelmiston elinkaarenhallinnan työkalu Polarion ALM opinnäytetyön toimeksiantajalle Valtra Oy Ab:lle. Käyttöönoton tavoitteena oli tukea Valtran ohjelmistokehitystä parantamalla ohjelmiston elinkaaren eri vaiheiden ja tuotosten jäljitettävyyttä toisiinsa sekä kykyä suunnitella ja seurata edistymistä. Polarionin käyttöönotto oli projektina ajankohtainen, koska Valtralla ei ollut aikaisemmin käytössä ohjelmiston elinkaarenhallinnan työkaluja ja Valtran useat yhtäaikaiset projektit, tuotteet ja niihin kehitettävät ohjelmistot muodostivat tarpeen hallita ohjelmiston elinkaaren eri vaiheita jäljitettävällä ja tehokkaalla tavalla.

Opinnäytetyön teoriaosassa käsitellään lyhyesti ohjelmiston elinkaarenhallinnan taustaa sekä Valtran ohjelmistokehityksen tilannetta ja erityispiirteitä ennen Polarionin käyttöönottoa. Valtran ohjelmistokehityksen tilannetta tarkastellaan ohjelmiston elinkaarenhallinnan näkökulmasta. Teorian ja Valtran menetelmien tarkastelu ennen käyttöönottoa toimivat perustana opinnäytetyön varsinaiselle toteutukselle, eli Polarionin käyttöönotolle ja mukautukselle. Käytännön toteutuksista kuvaillaan käyttöönoton vaiheita ja Polarionin mukauttamista tukemaan Valtran erityispiirteitä.

2 OHJELMISTON ELINKAARENHALLINTA

Ohjelmiston elinkaarella tarkoitetaan koko sitä ajanjaksoa, jolla ohjelmistoa kehittävä organisaatio käyttää siihen resursseja. Ohjelmiston elinkaari voidaan jakaa kolmeen osa-alueeseen, jotka ovat johtaminen, ohjelmistokehitys ja ohjelmiston ylläpito (ks. kuvio 1). Ajallisesti ohjelmiston elinkaaren tapahtumat on selkeää jakaa toimitusta edeltävään kehittämisvaiheeseen ja toimituksen jälkeiseen ylläpitovaiheeseen. (Chappel 2008, 2.)



KUVIO 1. Ohjelmiston elinkaari (Chappel 2008, 2, muokattu)

Koko ohjelmiston elinkaaren ajalle ulottuva johtaminen tarkoittaa päätöksentekoa ja projektinhallintaan liittyviä tehtäviä. Ohjelmistokehitys käsittää ohjelmiston suunnittelun ja käytännön toteuttamisen vaiheet. Tämä vaihe vie tyypillisesti paljon resursseja toimitusta edeltävässä kehittämisvaiheessa, mutta ilmenee usein vahvasti myös toimituksen jälkeisessä ylläpitovaiheessa. Ohjelmiston ylläpito tarkoittaa ohjelmiston julkaisua ja sen jälkeen tehtäviin muutoksiin ja korjauksiin liittyviä tehtäviä. (Chappel 2008, 3-5).

Ohjelmiston suunnittelun ja toteutuksen käsittävää ohjelmistokehitystä pidetään yleisesti tärkeimpänä yksittäisenä vaiheena ohjelmiston elinkaaressa. On kuitenkin tärkeää huomata myös johtamisen ja ylläpitovaiheen tapahtumien merkittävä rooli

onnistumisessa. Parhaan liiketoiminnallisen ja laadullisen tuloksen saavuttamiseksi kaikkien näiden kolmen osa-alueen tulisi toimia tiiviisti yhdessä. (Chappel 2008, 5). Onnistuminen vain yhdellä osa-alueella ei välttämättä takaa tuotetta, joka vastaa liiketoiminnallisia tarpeita ja vaatimuksia. (Schwaber 2006, 5).

Ohjelmiston elinkaarenhallinnalla tarkoitetaan ohjelmiston elinkaaren vaiheiden koordinointia ja elinkaaren eri vaiheissa syntyvien tuotosten hallintaa koko ohjelmiston elinkaaren ajan. (Schwaber 2006, 3.) Ohjelmiston elinkaarenhallinnan tarkoitus on auttaa selviytymään ohjelmistokehityksestä helpommin ennustettavalla ja kustannustehokkaalla tavalla (Shaw 2007, 4).

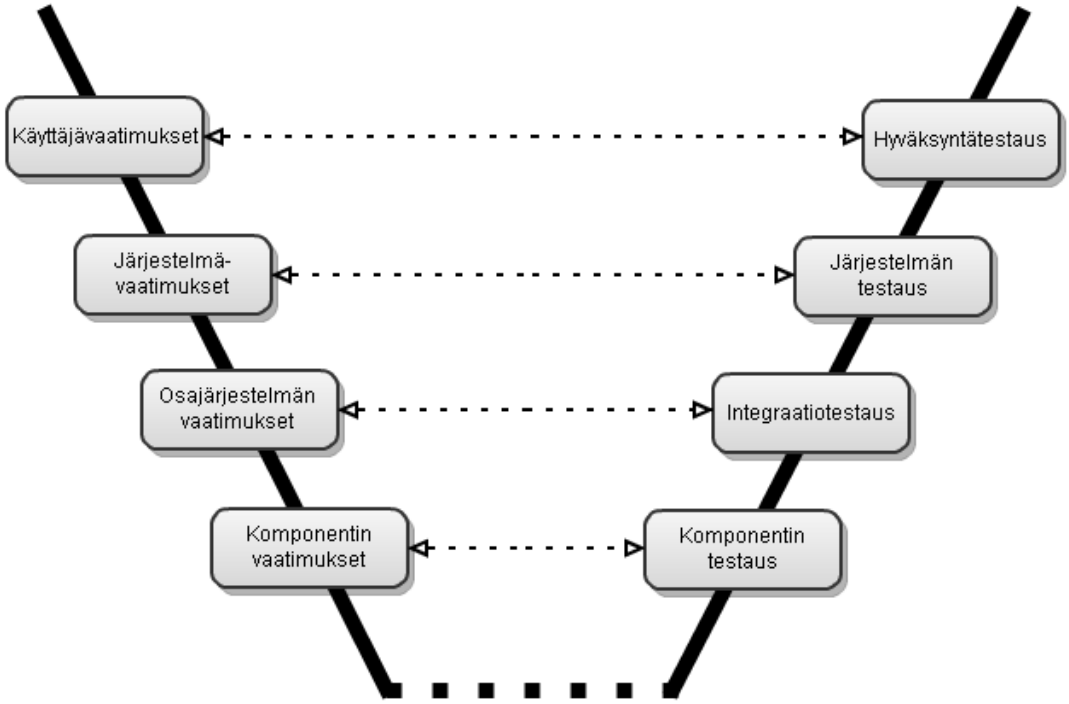
Ohjelmiston elinkaarenhallinnasta on kirjoitettu pääasiassa sitä tukevien menetelmien sekä työkalujen näkökulmista. Schwaber (2006, 4) määrittelee jäljitettävyyden, prosessien automatisoinnin ja raportoinnin olevan ohjelmiston elinkaarenhallinnan kolme tukipilaria. Kääriäinen (2011, 21) toteaa ohjelmiston elinkaarenhallinnan taustalla olevan vaatimustenhallinta ja jäljitettävyys, ohjelmiston konfiguraationhallinta sekä työkalujen integraatiot.

Ohjelmistokehityksen työkalut ovat kehittyneet tilanteeseen, jossa elinkaaren eri vaiheille on omat työkalut. Työkaluilla on usein omat formaatit, sisäisiä prosesseja ja asetuksia sekä mahdollisesti oma versiohallinnan arkisto. (Rizzo 2011, 8). Ohjelmiston elinkaarenhallinnan kannalta eri työkalujen tulisi muodostaa yhtenäinen kehitysympäristö, joka tukee koko ohjelmiston elinkaarta (Kääriäinen 2011, 25).

Kallio (2008, 41) toteaa, että ohjelmiston ja perinteisesti fyysisen tuotteen elinkaarenhallinnalla on yhteisiä tekijöitä. Kääriäisen (2011, 20) mukaan Stark (2005) määrittelee tuotteen elinkaarenhallinnan (*PLM*) kehittyneen tarpeesta tuoda yhteen toisistaan erilliset tuotteen kehityksen vaiheet ja prosessit, tarkoituksena välttää eri vaiheiden välille kadonnutta informaatiota, väärinymmärrettyjä vaatimuksia sekä perustelematonta päätöksentekoa.

Poiketen muista ohjelmiston elinkaaren vaiheista johtaminen ja vaatimustenhallinta ulottuvat koko elinkaaren ajalle (Shaw 2007, 4). Ohjelmiston elinkaaressa vaatimuk-

set ovat perustana mm. ohjelmiston suunnittelulle, riskien arvioimiselle, ohjelmiston testaukselle ja muutosten hallinnalle (Hull, Jackson & Dick 2005, 2). Vaatimusten jäljitettävyys toteutukseen ja testaukseen (ks. kuvio 2) mahdollistavat objektiivisen tavan seurata edistymistä (Hull ym. 2005, 3).



KUVIO 2. Käyttäjävaatimuksista hyväksyntätestaukseen (Hull ym. 2005, 7, muokattu)

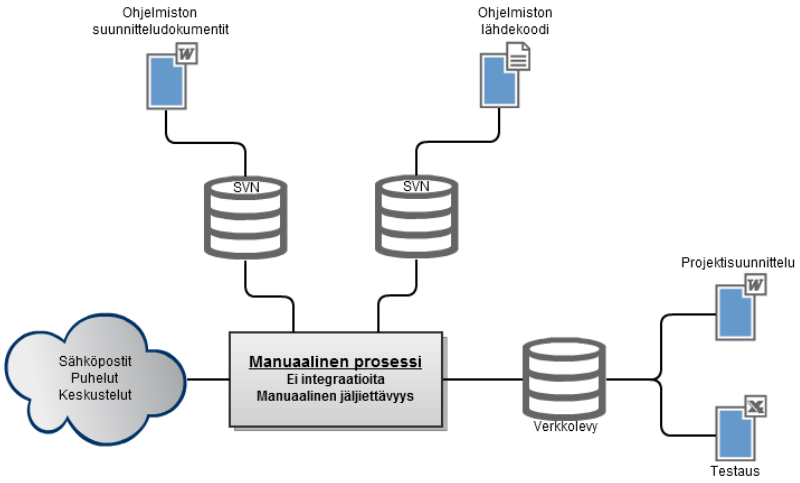
3 VALTRAN OHJELMISTOKEHITYKSEN NYKYTILANNE

3.1 Monimutkainen ympäristö ja hajautettu dokumentaatio

Valtralla kehitettävä ohjelmisto liittyy usein yhtäaikaisesti useaan eri projektiin, tuotteeseen ja toimintoon traktorissa. Uusien tuotteiden ja toimintojen kehittämisen lisäksi resursseja kuluu myös tuotannossa olevien tuotteiden ylläpitämiseen. Kehitettävä ohjelmisto muodostaa yhdessä taustalla toimivan sähköjärjestelmän ja mekaniikan kanssa laajan sulautetun järjestelmän.

Johtamisen kannalta useat rinnakkaiset projektit ja tuotteet aikatauluineen aiheuttavat haasteita suunnittelussa ja päätöksenteossa. Ohjelmistokehityksen näkökulmasta useat rinnakkain kehitettävät ohjausyksiköt, toiminnot ja niiden riippuvuudet toisiinsa koettelevat kykyä hallita ohjelmistokehitystä järjestelmällisesti. Kehittämisen lisäksi monimutkaisen sulautetun järjestelmän testaaminen on oma haasteensa. Samat haasteet heijastuvat myös ohjelmiston ylläpitovaiheessa tehtäviin muutoksiin ja vikakorjauksiin.

Useat eri ohjelmistokehityksen aktiviteetit toimivat Valtralla omina kokonaisuuksinaan hyvin, mutta ohjelmiston elinkaarenhallinnan kannalta tärkeät jäljitettävyys, edistymisen seuranta ja raportointi sekä työkalujen integraatiot tuottavat haasteita. Pääasiallinen syy tähän on se, että ohjelmiston elinkaaren tuotokset, kuten vaatimukset, testitapaukset, havaitut viat, muutospyynnöt jne. eivät ole nykytilanteessa suoraan jäljitettävissä toisiinsa, vaan ne ovat dokumentoituina hajautetusti eri formaatteihin (ks. kuvio 3).



KUVIO 3. Valtran ympäristö ennen ALM-työkalua

Vaikka ohjelmiston elinkaarenhallinta ei keskity yksittäisiin elinkaaren aktiviteetteihin, on luvussa 4 esiteltyjen ratkaisujen ymmärtämisen kannalta hyödyllistä esitellä ensin käytännön haasteita eri elinkaaren vaiheista. Luvuissa 3.2 – 3.5 käsitellään vaatimustenhallinnan, vikojen seurannan, testauksen ja ohjelmiston julkaisujen tilannetta ennen ALM-työkalun käyttöönottoa.

3.2 Vaatimustenhallinta

Valtran tuotehallinta määrittelee tuotteen liiketoiminnalliset, standardeihin ja määräyksiin liittyvät sekä osittain myös käyttäjän vaatimukset. Näiden pohjalta sähkö-, automaatio- ja ohjelmistosuunnittelusta vastaava ryhmä määrittelee järjestelmän vaatimukset ja suunnittelee vaatimuksien mukaisen toteutuksen. Järjestelmän vaatimukset dokumentoidaan osaksi traktorin toimintokuvauksia ja muuta suunnitteludokumentaatiota.

Traktorin perusominaisuuksiin liittyvät vaatimukset ovat traktorin mallista riippumatta osittain samoja. Toisistaan poikkeavat toiminnot ovat dokumentoituna toimintokohtaisiin variaatioihin, jolloin joukko vaatimuksia toteutuu kaikissa malleissa ja osa vaatimuksista vain tietyissä malleissa. Kaikille malleille yhteiset vaatimukset ja niiden eri variaatiot muodostavat monimutkaisen ympäristön, jossa vaatimusten toteutumista ja testausta eri traktorimalleissa on vaikea seurata. Koska vaatimukset ovat dokumentoitu toimintokohtaisiin dokumentteihin, yksittäisten vaatimusten jäljittäminen kokonaisten toimintojen joukosta on haasteellista. Traktorin ominaisuuksien

testaaminen perustuu vaatimusten pohjalta laadittuun joukkoon testitapauksia. Suoritettujen testien tuloksien jäljittäminen niitä vastaaviin vaatimuksiin on nykyisin menetelmin haasteellista.

3.3 Vikojen seuranta

Ohjelmistoon liittyviä vikoja voivat havaita useat eri osapuolet. Vikoja voivat havaita esimerkiksi ohjelmistoon tehtyä muutosta testaavat suunnittelijat tai kehityksen aikaista testijulkaisua testaavat koeajokuljettajat. Toisinaan vikoja havaitaan myös elinkaaren ylläpitovaiheessa, jolloin sen ilmoittajana voi olla esimerkiksi huolto-organisaatio.

Vikakuvaukset saapuvat suunnittelijoiden tietoon useissa eri formaateissa, joista yleisimpinä koeajokuljettajien Excel- ja Word -raportit sekä kommunikoinnin välineet, kuten sähköposti, puhelut tai keskustelut. Usein vian kuvaus toimii alkuperäisessä formaatissaan dokumentaationa vian korjaamiseen saakka.

Suunnittelijat arvioivat havaittujen vikojen vakavuutta ja mahdollisesti niistä kuljettajalle tai traktorille aiheutuvia riskejä. Riippuen vikojen vakavuudesta ja niiden aiheuttamista riskeistä, vikojen korjaamisesta ja korjatun ohjelmaversion jakelusta sovitaan suunnittelijoiden kesken tai viikoittaisessa ohjelmistokehitykseen liittyvässä palaverissa.

Vikojen seuranta on nykyisin menetelmin hankalaa, koska eri lähteistä saapuvien vikakuvausten kirjaamiseen ja hallintaan ei ole yhtenäistä formaattia tai järjestelmää. Valtralla oli koekäytössä vikojen seurantaan Bugzilla, mutta sen käyttö jäi kuitenkin vähäiseksi.

3.4 Ohjelmiston testaus

Valtralla ohjelmistoa testataan useilla eri tavoilla. Traktorissa ohjelmistoa testaavat muutoksia tehneet suunnittelijat sekä koeajokuljettajat. Tämän lisäksi kehitettäviä traktoreita ja niiden osia testataan laboratorioissa sekä todellisissa työympäristöissä.

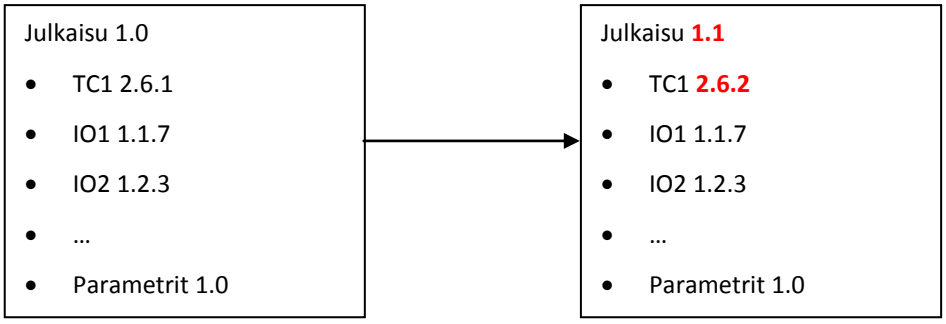
Koeajokuljettajien suorittamat testitapaukset ovat määriteltyinä Excel-dokumenttiin, johon testitulokset ja havaitut ongelmat dokumentoidaan testauksen aikana. Koeajokuljettajat raportoivat havaituista ongelmista suunnittelijoille ja koeajoraportin tuloksia käydään läpi viikoittaisessa palaverissa.

Pääasiallinen ympäristö ohjelmiston testaukselle on traktori, mutta sen lisäksi mm. vaihteisto-ohjaimen toimintaa testataan myös HIL-ympäristössä (Hardware-In-Loop). HIL-ympäristössä elektroninen ohjausyksikkö voidaan kytkeä osaksi testauslaitteistoa, joka tuottaa ohjausyksikön sisään- ja ulostuloihin (I/O) sähköisesti todellista vastaavan ympäristön. Valtralla HIL-ympäristössä testataan traktorin voimansiirron ja ajologiikan toimintoja Simulink-mallin avulla. HIL-ympäristössä suoritetaan manuaalisten testien ja mittausten lisäksi myös automatisoituja testejä. Automatisoitujen testien tulokset ovat saatavilla HTML- tai PDF-raportin muodossa, ja ne voidaan lähettää suunnittelijoiden sähköpostiin.

Muiden elinkaaren vaiheiden tavoin testaus toimii eri muodoissaan hyvin, mutta testitapaukset ja tulokset eivät ole helposti hallittavissa tai jäljitettävissä ohjelmiston vaatimuksiin. Ohjelmiston elinkaarenhallinnan kannalta ohjelmiston testauksen tulisi olla tiiviisti yhteydessä mm. ohjelmiston vaatimuksiin.

3.5 Ohjelmiston jakelut

Ohjelmistosta julkaistavat paketit muodostuvat eri ohjausyksiköiden ohjelmaversioista ja parametreista. Eri ohjausyksiköiden ohjelmaversioita kehitetään tarpeiden mukaan, ja ne muodostavat yhdessä parametrien kanssa traktoriin ladattavan konfiguraation. Myös julkaistavilla paketeilla on oma versiointi, joka kehittyy yhdenkin konfiguraation osan muuttuessa (ks. kuvio 4).



KUVIO 4. Julkaisujen versiointi

Kun ohjelmisto täyttää julkaisua edellyttävät kriteerit, se jaetaan tuotetiedon hallintajärjestelmään (PDM) osaksi tuotetta. Jakelun jälkeen se on tuotannon ja huoltoorganisaation käytettävissä ja se voidaan ladata huoltotyökalulla traktoriin. Julkaistavaan pakettiin liitetään dokumentaatio, jossa kuvataan mm. mahdolliset uudet ominaisuudet, muutokset ja korjatut ongelmat. Ohjelmiston julkaisujen ajankohdan määrittävät kehitettävän traktorin projektisuunnitelmaan kuuluvien prototyyppien valmistuminen sekä sarjatuotannon aloittaminen.

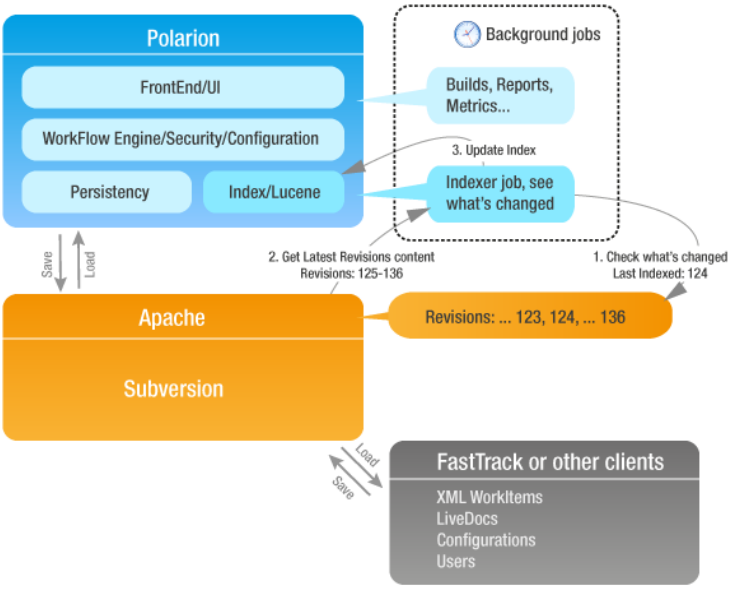
Ohjelmistojen jakelu on havaittu toimivaksi, mutta jakelun mukaan liitettävän dokumentaation laatiminen on haasteellista. Ohjelmistoon toteutettuja kehitys- ja muutospyyntöjä, vaatimuksia tai vikakorjauksia ei ole saatavilla helposti, vaan tiedot kerätään ”manuaalisesti” pääasiassa suunnittelijoilta.

4 POLARION ALM OHJELMISTOKEHITYKSEN TUKENA

4.1 Polarion ALM

Polarion Softwaren kehittämillä *Polarion Requirements Management* ja *Polarion ALM* -ohjelmistoilla on maailmanlaajuisesti noin miljoona käyttäjää. Polarion tarjoaa yhtenäisen ja kustannustehokkaan web-pohjaisen ratkaisun ohjelmiston elinkaarenhallintaan. Polarion tuo ominaisuuksillaan ohjelmistokehitykseen jäljitettävyyttä sekä läpinäkyvyyttä ja tukee kaikkia ohjelmistokehityksen tärkeimpiä aktiviteetteja. (About Polarion Software n.d).

Polarionin arkkitehtuurista (ks. kuvio 5) merkittävä osa perustuu avoimen lähdekoodin sovelluskehyksiin ja rajapintoihin. Polarionissa kaikki tiedot ja asetukset ovat tallennettuna xml-muodossa Subversion versiohallinnan arkistoon. Taustalla toimivan Subversion arkiston ja Apache palvelinohjelmiston päällä toimii sivumoottori XWiki. Arkistoituun tietoon kohdistuvia hakuja ja taustalla toimivaa tiedon indeksointia toteuttaa Apache Lucene-hakumoottori. Wiki-sivuille voidaan luoda dynaamista sisältöä XWikin avoimia Java-rajapintoja käyttävien Apache Velocity-makrojen avulla. Wiki-sivuihin voidaan sisällyttää myös mm. HTML-, CSS- tai Javascript-osioita. Hyvin dokumentoitu ja avoimiin teknologioihin perustuva arkkitehtuuri mahdollistaa Polarionin laajan muokkaamisen.



KUVIO 5. Polarion arkkitehtuuri (Polarion Performance & Scalability 2010)

4.2 Käyttöönotto

Polarionin käyttöönotto aloitettiin hankkimalla ilmainen kuukauden koekäyttöön tarkoitettu lisenssi. Polarion asennettiin aluksi testiympäristöön, jossa arvioitiin Polarionin ominaisuuksia ohjelmiston elinkaaren eri vaiheiden hallintaan. Kokeilujen tuloksia esiteltiin suunnittelijoille viikoittaisissa palavereissa ja ominaisuuksia otettiin käyttöön vaiheittain. Käyttöönotossa edettiin vaiheittain, koska suunnittelijoita ei haluttu kuormittaa liikaa uusilla tavoilla/työkalulla dokumentoida ja hallita ohjelmiston elinkaaren eri vaiheita.

Lähtökohtana uusien ominaisuuksien käyttöönotossa oli säilyttää jo kehitettyyn ohjelmistoon liittyvät dokumentaatiot sellaisenaan ja keskittyä dokumentoimaan Polarioniin uusia kehitys- ja muutospyyntöjä, vaatimuksia, tehtäviä, havaittuja vikoja ja testitapauksia. Ylläpidettävien tai lähes valmiiden tuotteiden vaatimusten ja muiden tuotosten kerääminen traktorin toimintokuvausten ja muiden teknisten suunnitteludokumenttien joukosta olisi vaatinut kohtuuttoman paljon resursseja eikä siksi ollut perusteltua. Alkuvaiheessa Polarionia käytettiin pääasiassa projekteihin ja tuotteisiin liittyvien kehitys- ja muutospyyntöjen sekä havaittujen vikojen hallintaan ja seurantaan. Myöhemmin Polarioniin dokumentoitiin myös vaatimuksia ja testitapauksia.

Polarionin mukauttaminen tukemaan Valtran ohjelmistokehityksen tarpeita ja erityispiirteitä oli keskeisessä osassa käyttöönottoa ja siihen käytettiin opinnäytetyön aikana eniten resursseja. Suunnittelijoiden toiveiden ja tarpeiden mukaisia kokeiluja tehtiin testiympäristössä koko opinnäytetyön ajan. Suunnittelijoiden palautteen perusteella hyväksi todettuja mukautuksia otettiin käyttöön varsinaiseen ”tuotantoympäristöön”.

4.3 Mukauttaminen

Polarion mukautuu hyvin erilaisiin ohjelmistokehityksen prosesseihin. Asennuksen mukana tuli joukko valmiiksi määriteltyjä projektipohjia ja asetuksia, mutta ne eivät soveltuneet sellaisenaan Valtran käyttöön. Määriteltäviä asetuksia oli paljon, mutta

niistä opinnäytetyön kannalta tärkeimpinä on esitelty ohjelmiston elinkaaren eri tapahtumia tai tuotoksia kuvaavat tietotyypit, tietotyypeille määritellyt kentät sekä työnkulut.

Polarionin tietotyypit mukautettiin vastaamaan Valtran ohjelmiston elinkaaren tärkeitä tapahtumia/tuotoksia. Tietotyyppien ja niiden työnkulkujen määrittely mahdollistaa Polarionin käytön monipuolisesti erilaisiin tarkoituksiin. Tietotyypillä voidaan kuvata esimerkiksi yksinkertaista työtehtävää, jolla on vain kaksi mahdollista tilaa ja vain oletuksena määritellyt kentät. Toisaalta tietotyyppi voi sisältää paljon mukautettuja kenttiä ja kymmeniä tiloja käsittävän työnkulun, joka vaatii useiden eri käyttäjien toimenpiteitä edistyäkseen. Lähtökohtana tietotyyppien määrittelyssä Valtran ympäristöön oli määritellä vain olennaisimmat tiedot ja säilyttää työnkulku suoraviivaisena ja helposti seurattavana. Tietotyypit määriteltiin taulukon 1 mukaisesti.

TAULUKKO 1. Määritellyt tietotyypit

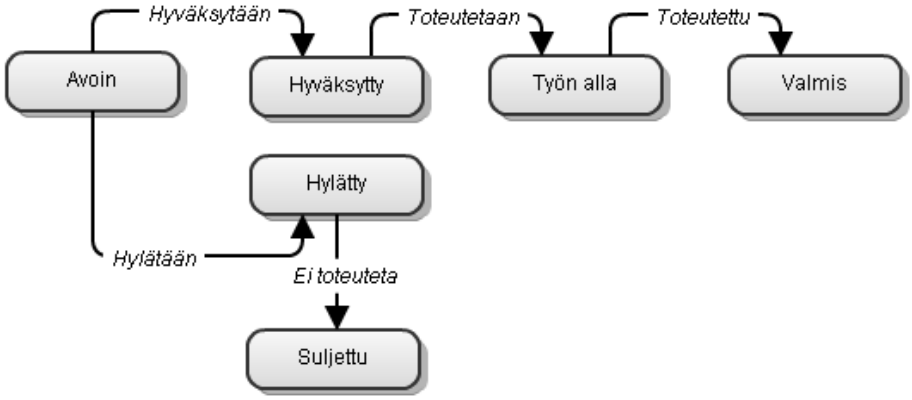
Tietotyypit Polarionissa	**Kuvaus**
Request	Kehitys- tai muutospyyntö
Requirement	Vaatimus
Task	Tehtävä, johon käytetään resursseja
Test Case	Ohjelmiston testitapaus
Defect	Ohjelmistossa havaittu vika
Release	Ohjausyksikön ohjelmaversio
Package	Julkaistava ohjelmistopaketti

Ohjelmiston elinkaaren eri tapahtumat tai tuotokset voivat liittyä Valtralla yhtäaikaisesti projekteihin, tuotteisiin ja toimintoihin. Tämän vuoksi kaikille tietotyypeille määriteltiin kentät *Project, Product* ja *Fuction*. Yhteisten kenttien lisäksi tietotyypeille määriteltiin myös muita Valtran ympäristöön liittyviä kenttiä.

Kehitys- ja muutospyynnöt

Kehitettävään tai ylläpidettävään ohjelmistoon liittyviä kehitys- ja muutospyyntöjä voi saapua useissa eri formaateissa eri osapuolien toimesta. Yhtenäinen tapa käsitellä ja dokumentoida pyyntöjä oli tärkeää, joten tähän tarkoitukseen määriteltiin tietotyyppi *Request.* Tietotyypille määriteltiin *Source*-kenttä, joka kuvaa pyynnön lähdettä. Valtran tapauksessa lähteenä voi olla esimerkiksi tuotehallinta tai huolto-

organisaatio. Kehitys- ja muutospyyntöjen työnkulku määriteltiin kuvion 6 mukaisesti.



KUVIO 6. Kehitys- ja muutospyynnön työnkulku

Vaatimukset

Kehitettävän ohjelmiston tulee toimia määriteltyjen liiketoiminnallisten, käyttäjä- sekä järjestelmävaatimusten mukaisesti. Vaatimustenhallinnan tueksi määriteltiin tietotyyppi *Requirement*. Vaatimukselle määriteltiin samanlainen työnkulku kehitys- ja muutospyyntöjen kanssa (ks. kuvio 6).

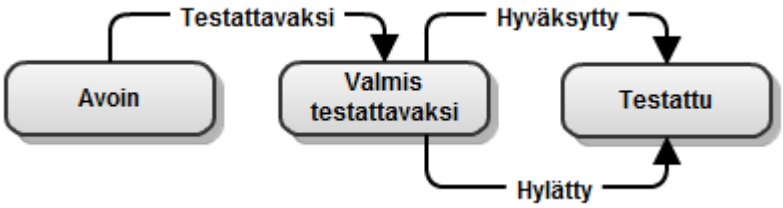
Tehtävät

Polarionissa tehtävillä tarkoitetaan suunnittelijoiden resursseja kuluttavaa työtä. Tyypillisesti tehtävät liittyvät muihin Polarioniin tallennettuihin tuotoksiin, kuten kehitys- ja muutospyyntöihin, vaatimuksiin tai havaittuihin vikoihin. Tehtäviä kuvattiin tietotyypillä *Task*. Tehtävien työnkulkuun määriteltiin vain kolme vaihetta, jotka olivat avoin, työn alla ja valmis (ks. kuvio 7).

KUVIO 7. Tehtävän työnkulku

Testitapaukset

Ohjelmiston testitapauksia määriteltiin aikaisemmin mm. Excel-dokumentteihin ja HIL-ympäristön automatisoitujen testien formaatteihin. Polarioniin määriteltiin tietotyyppi *Test Case* kuvaamaan kaikkia ohjelmiston testitapauksia. Manuaaliset, hyväksyntä- ja HIL-testitapaukset eroteltiin toisistaan määrittelemällä *Test Type*-kenttä. Testitapauksen työnkulku määriteltiin kuvion 8 mukaisesti.



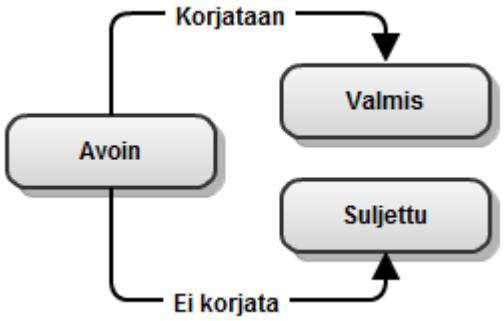
KUVIO 8. Testitapauksen työnkulku

Viat

Ohjelmistossa havaittuja vikoja kuvattiin tietotyypillä *defect*. Polarionin oletuksena määriteltyihin kenttiin kuului vakavuutta kuvaava *Severity*. Oletuksena vakavuutta arvioitiin numeroasteikolla 0-100. Tällä asteikolla havaitun vian vakavuuden arviointi ei ollut riittävän kuvaava, joten vakavuuden arviointiin luotiin taulukon 2 mukainen asteikko. Vikojen työnkulku määriteltiin kuvion 9 mukaisesti.

TAULUKKO 2. Määritelty asteikko vakavuuden arviointiin

Vakavuus	**Kuvaus**
Safety Critical	Vika voi aiheuttaa vaaraa kuljettajan terveydelle
Machine Critical	Vika voi aiheuttaa vaaraa ajoneuvolle
Not Acceptable	Ajoneuvo ei toimi hyväksyttävällä tavalla
Annoying	Vika on ärsyttävä
Minor	Vika on vähäinen



KUVIO 9. Vian työnkulku

Ohjelmiston julkaisut

Ohjelmistosta julkaistavaa pakettia kuvattiin tietotyypillä *Package* ja ohjausyksikön ohjelmaversiota tietotyypillä *Release*. Ohjelmiston julkaisuihin liittyvien tietotyyppien pääasiallinen tarkoitus oli tukea julkaisuun liitettävän dokumentaation laatimista. Ohjausyksikköihin tehtyjä vikakorjauksia ja muutoksia liitettiin Polarionissa ohjausyksiköiden ohjelmaversioihin (*Release),* ja yhdessä ne muodostivat ohjelmistosta julkaistavan paketin (*Package).* Polarionissa julkaisujen työnkulku oli yksinkertainen ja molemmilla julkaisuihin liittyvillä tietotyypeillä samanlainen (ks. kuvio 10).

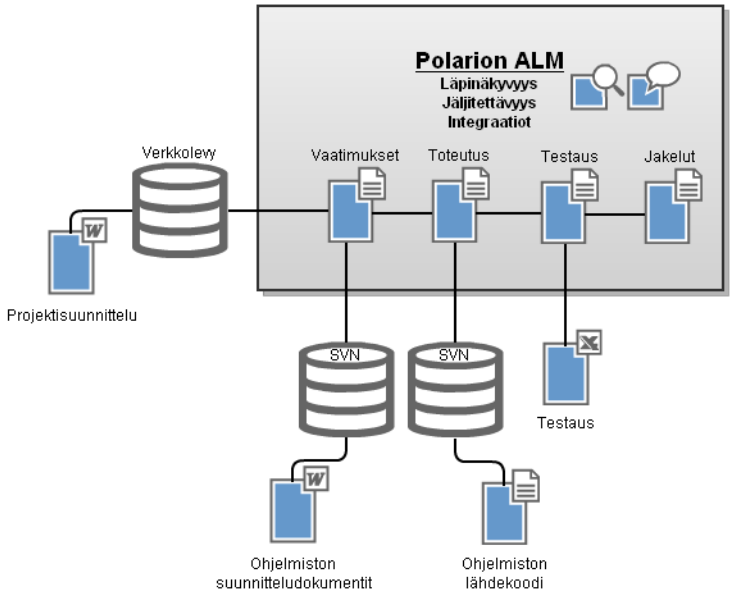
KUVIO 10. Ohjelmiston julkaisun työnkulku

4.4 Jäljitettävyyttä ohjelmiston elinkaareen

4.4.1 Polarion ohjelmiston elinkaaressa

Jäljitettävyyden kannalta Polarionin käyttöönotto tähtäsi ympäristöön, jossa ohjelmiston elinkaaren eri vaiheita ja tapahtumia voidaan jäljittää toisiinsa (ks. kuvio 11). Jäljitettävyyden perustana toimivat tietotyyppien yhteydet ja suhteet toisiinsa. Polarionissa jokaiselle tietotyypille määriteltiin sallitut suhteet muihin tietotyyppeihin.

Esimerkiksi vaatimukseen liitettäviä tietotyyppejä voivat olla mm. toteuttava tehtävä (*implementing task)* tai vahvistava testitapaus (*verifying testcase)*.

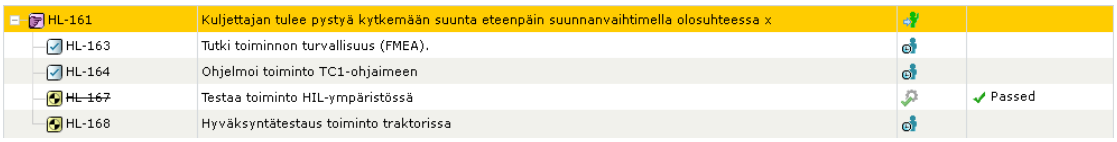


KUVIO 11. Polarion ohjelmiston elinkaaressa

Tietotyyppeihin voi liittyä myös Polarionin ulkoisia tuotoksia, kuten suunnitteludokumentaatiota, lähdekoodia tai testitapauksia. Polarioniin voi määritellä ulkoisiksi lähteiksi Subversion- tai Git-versiohallinnan arkistoja, joihin tehtyjä muutoksia voidaan näyttää suoraan Polarionissa. Valtralla suunnitteludokumentaatio ja lähdekoodi sijaitsevat Subversion-arkistoissa. Tämä mahdollistaa esimerkiksi lähdekoodiin tehdyn muutoksen näkyvyyden Polarionissa tallennettuun tehtävään.

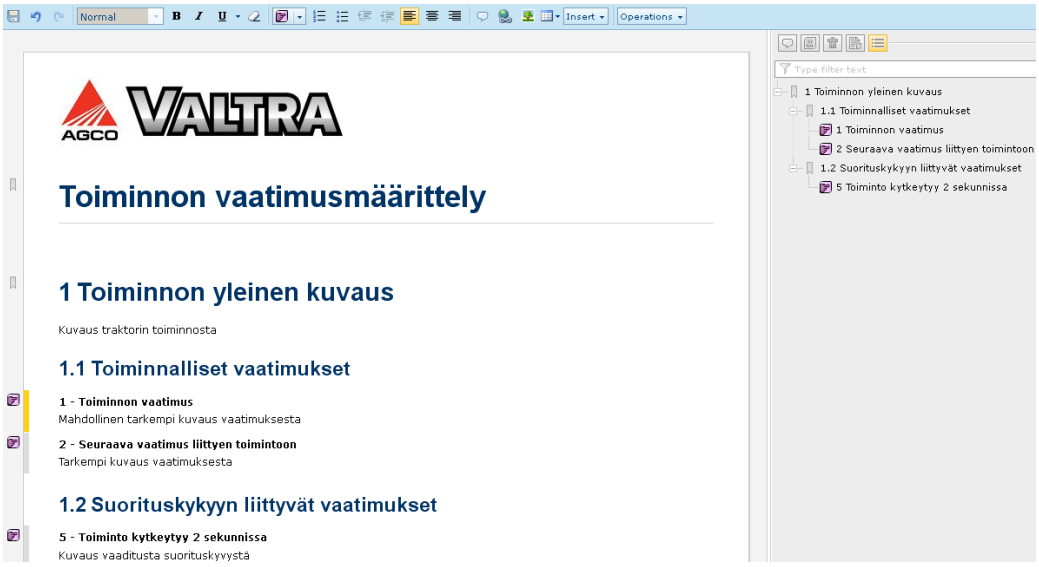
4.4.2 Vaatimuksista toteutukseen

Vaatimukset toimivat perustana monelle eri tapahtumalle ohjelmiston elinkaaressa. Polarionissa vaatimuksiin liitettiin tyypillisesti toteuttavat tehtävät sekä vahvistavat testitapaukset (ks. kuvio 12). Vaatimusten, tehtävien ja testitapausten liittäminen toisiinsa muodostaa perustan vaatimusten edistymisen seurannalle ja jäljitettävyydelle. Vaatimusten ja testitapausten väliset suhteet ovat tärkeitä myös tarkastellessa ohjelmiston testauksen kattavuutta, eli tarkastellessa kuinka suuri osa vaatimuksista vahvistetaan testitapauksella.



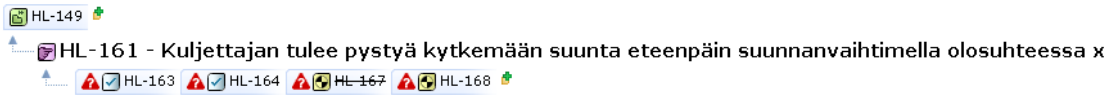
KUVIO 12. Vaatimuksen yhteys toteutukseen ja testaukseen

Valtralla vaatimukset olivat tyypillisesti Word-dokumenttien muodossa, jolloin yksittäisten vaatimusten jäljittäminen oli haasteellista. Polarionissa vaatimuksia tai muita tietotyyppejä voi muokata dokumenttinäkymässä (ks. kuvio 13), mutta höytyä silti niiden jäljitettävyydestä.



KUVIO 13. Vaatimusten määrittely dokumenttinäkymässä

Vaatimukset voivat muuttua esimerkiksi ohjelmiston julkaisun jälkeisessä ylläpitovaiheessa. Polarion voidaan konfiguroida huomauttamaan muuttuvien vaatimusten vaikutuksista. Kuvion 14 tapauksessa vaatimukseen oli liitetty suunnitteluun ja toteutukseen liittyviä tehtäviä sekä testitapauksia. Tilanteessa, jossa useat suunnittelijat toteuttivat samaan vaatimukseen liittyviä tehtäviä tai testitapauksia, oli hyödyllistä saada ilmoitus muutoksesta.

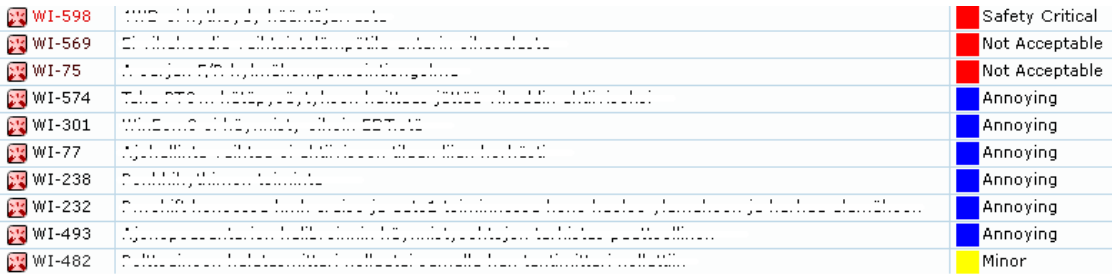


KUVIO 14. Muuttuneen vaatimuksen vaikutukset korostettuna

4.4.3 Vikojen seuranta

Vikojen seuraamisen kannalta tärkeitä tietoja olivat vakavuus sekä havainnon ja suunnitellun korjauksen ohjelmaversiot. Määriteltyjen tietojen perusteella vikoja voitiin seurata ja jäljittää eri perusteilla ja tehdä päätöksiä korjauksesta mm. vakavuuden perusteella (ks. kuvio 16). Havaitut viat pyrittiin liittämään myös osaksi toiminnon kuvaavia vaatimuksia, mikäli ne olivat dokumentoituna Polarioniin (ks. kuvio 15).

KUVIO 15. Havaittu vika liitettynä vaatimukseen



KUVIO 16. Viat listattu kriittisyyden perusteella

4.4.4 Ohjelmiston julkaisut

Polarionissa ohjelmiston julkaisuihin liitettiin mm. kehitys- ja muutospyyntöjä, vaatimuksia ja korjattuja vikoja. Aikaisemmin ohjelmiston julkaisuun liitettävän dokumentaation tiedot kerättiin eri formaateista ja lähteistä. Polarionissa ohjelmiston julkaisuun liitettyjen tietojen puunäkymän (ks. kuvio 17) vienti Excel-raporttiin oli tärkeä ominaisuus julkaisujen dokumentoinnin kannalta (ks. kuvio 18).

TC1-2	TC1 -ohjaimen versio 2.8.0
~~TC1-3~~	<Kuljettajan> tulee pystyä suorittamaan <toiminto> <määrätyssä ajassa>
~~TC1-4~~	Tutki <toiminnon> soveltuvuus järjestelmään
~~TC1-5~~	Ohjelmoi <toiminto> TC1-ohjaimeen
~~TC1-6~~	<toiminto> testaus HIL-ympäristössä
~~TC1-7~~	<toiminto> hyväksyntätestaus traktorissa
~~TC1-8~~	<Huoltomiehen> tulee pystyä suorittamaan <toiminto> traktorin ollessa <tilassa>
~~TC1-9~~	Tutki onko <tila> turvallinen <toiminnolle>
~~TC1-10~~	Ohjelmoi tarvittava muutos TC1-ohjaimeen
~~TC1-11~~	<toiminto> testaus HIL-ympäristössä
~~TC1-12~~	<toiminto> hyväksyntätestaus traktorissa
~~TC1-13~~	<Huoltomies> ei pääse <huoltotilassa> suorittamaan <toimintoa>

KUVIO 17. Toteutuneet vaatimukset julkaisuun

Microsoft Excel - Xl0000019 [Read-Only]

File Edit View Insert Format Tools Data Window Help

Arial 11 B I U 85%

G32

	A	B	C	D
1	ID	Title	Type	Resolution
2	TC1-2	TC1 -ohjaimen versio 2.8.0	Release	
3	TC1-3	<Kuljettajan> tulee pystyä suorittamaan <toiminto> <määrätyssä ajassa>	Requirement	Implemented
4	TC1-4	Tutki <toiminnon> soveltuvuus järjestelmään	Task	Implemented
5	TC1-5	Ohjelmoi <toiminto> TC1-ohjaimeen	Task	Implemented
6	TC1-6	<toiminto> testaus HIL-ympäristössä	Test Case	Passed
7	TC1-7	<toiminto> hyväksyntätestaus traktorissa	Test Case	Passed
8	TC1-8	<Huoltomiehen> tulee pystyä suorittamaan <toiminto> traktorin ollessa <tilassa>	Requirement	Implemented
9	TC1-9	Tutki onko <tila> turvallinen <toiminnolle>	Task	Implemented
10	TC1-10	Ohjelmoi tarvittava muutos TC1-ohjaimeen	Task	Implemented
11	TC1-11	<toiminto> testaus HIL-ympäristössä	Test Case	Passed
12	TC1-12	<toiminto> hyväksyntätestaus traktorissa	Test Case	Passed
13	TC1-13	<Huoltomies> ei pääse <huoltotilassa> suorittamaan <toimintoa>	Defect	Fixed

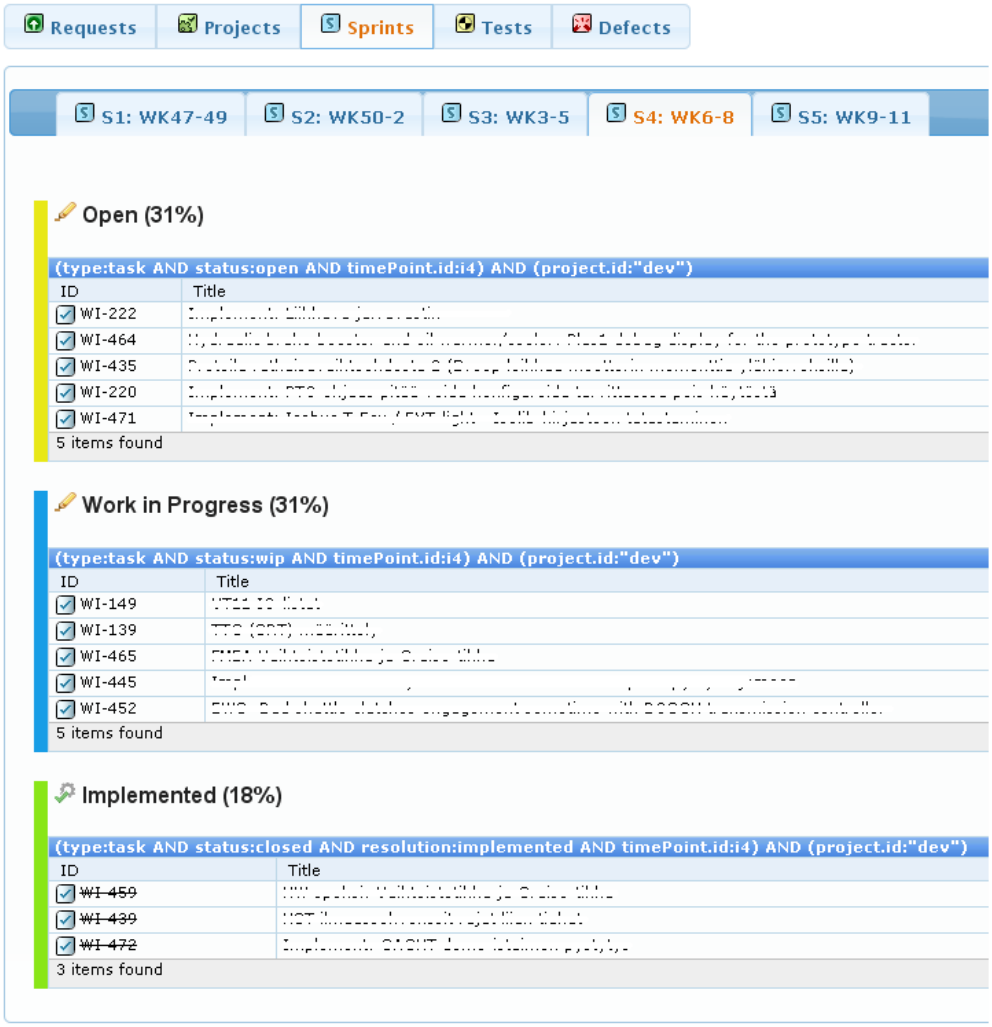
KUVIO 18. Excel-raportti julkaisun sisällöstä

4.5 Raportointi

Ohjelmistokehitykseen liittyvän päätöksenteon ja edistymisen seurannan kannalta raportointi edistymisestä on tärkeä osa ALM-työkalun toimintoja. Polarion tarjosi monipuolisia vaihtoehtoja erilaisten näkymien luomiseen. Raportoinnissa ja edistymisen seurannassa keskeisessä asemassa olivat Polarionin Wiki-sivut, joihin koottiin tietoja erilaisilla hakuehdoilla.

Polarionin Wiki-sivujen toteutus koostui tietojen hakemisesta (*Lucene / Velocity*), tietojen käsittelystä (*Velocity)* ja niiden esittämisestä käyttöliittymässä (*HTML, CSS ja Javascript)*. Wiki-sivujen näkymissä pyrittiin yhtenäiseen ulkoasuun ja helppoon käytettävyyteen. Käyttöliittymää laajennettiin jQueryUI Javascript-kirjaston komponenteilla, joilla tietoja voitiin näyttää/piilottaa ja selata välilehtien avulla.

Valtra käyttää ohjelmistokehityksessä ketterää SCRUM-prosessia, jossa ohjelmistoa kehitetään ja julkaistaan lyhyissä ja suunnitelluissa pyrähdyksissä. Edistymisen seurantaan kehitettiin näkymä, jossa pyrähdyksiin suunniteltujen tehtävien (*Task)* edistymistä voitiin seurata tehtävien tilan perusteella (ks. kuvio 19).

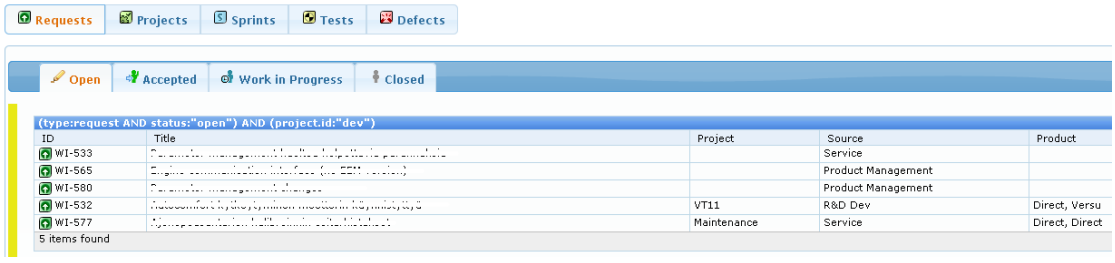


KUVIO 19. Näkymä pyrähdyksen tehtävien seurantaan

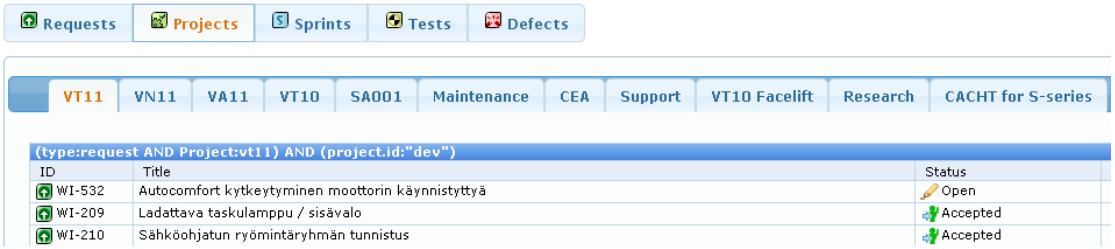
Ohjelmiston vaatimuksiin liitettiin usein myös mm. vaatimuksen toteuttavia tehtäviä sekä toiminnan vahvistavia testitapauksia. Oletuksena mahdollisten puunäkymien lisäksi vaatimusten edistymistä toteutukseen ja testaukseen haluttiin seurata kattavammin, joten tähän tarkoitukseen kehitettiin kuvion 20 mukainen näkymä. Tyypillisesti SCRUM-prosessissa seurataan edistymistä kuvion 20 tyyppisissä näkymissä.

KUVIO 20. Vaatimuksen edistymisen seuranta Wiki-näkymässä

Alkuvaiheessa Polarionia käytettiin eri lähteistä saapuvien kehitys- ja muutospyyntöjen käsittelyssä. Kehitys- ja muutospyyntöjä haluttiin seurata niiden tilan (ks. kuvio 21) tai projektien (ks. kuvio 22) perusteella. Käyttöliittymään kehitettyjen painikkeiden ja välilehtien hyödyt nousivat esille hyvin, kun samalle wiki-sivulle haettiin satoja tuotoksia eri perusteilla. Tietojen lajittelu ja esittäminen loogisissa kokonaisuuksissa helpotti käyttöä huomattavasti.



KUVIO 21. Kehitys- ja muutospyynnöt tilojen mukaan



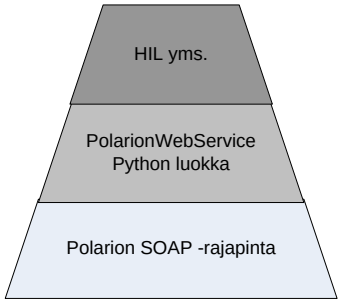
KUVIO 22. Kehitys- ja muutospyynnöt projektien mukaan

4.6 Laajennettavuus ja työkalujen integraatiot

Polarion on avoimen lähdekoodin ratkaisujen ja hyvin dokumentoitujen avointen rajapintojen ansiosta hyvin laajennettavissa. Ohjelmistoon on ladattavissa tuotteen kotisivujen kautta yli 100 valmista lisäosaa ja Polarionin käyttäjien työpanos avoimien lisäosien kehittämisessä on merkittävä. Osa laajoista lisäosista, kuten integraatiot muiden toimittajien kehitystyökaluihin, on kuitenkin kaupallisia. Valmiiden lisäosien lisäksi Polarion tarjoaa esimerkkejä ja ilmaiset työkalut omien lisäosien kehittämiseen.

SOAP on yksinkertainen xml-pohjainen protokolla järjestelmien väliseen tiedonsiirtoon, jossa tiedot lähetetään kaikkien selainten ja palvelinten tukemaa HTTP-protokollaa käyttäen. SOAP mahdollistaa alustariippumattoman tavan kommunikoida Internetin välityksellä. (SOAP Introduction n.d.)

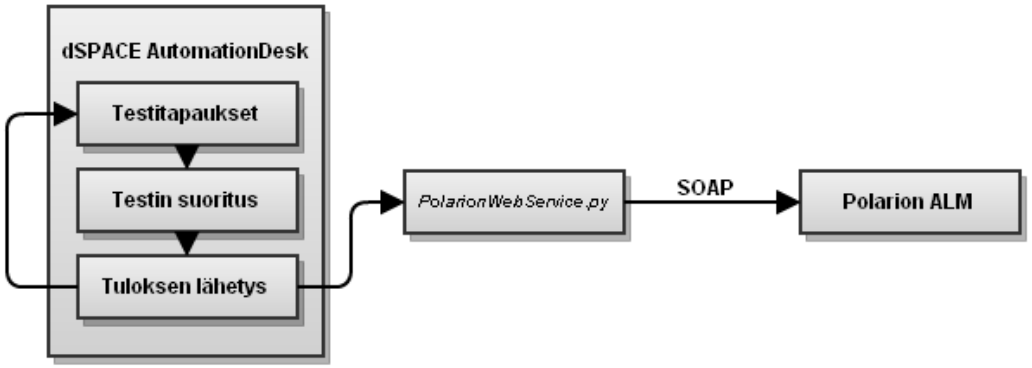
Laajennettavuuden ja työkalujen integraatioiden osalta opinnäytetyössä perehdyttiin Polarionin SOAP-rajapinnan hyödyntämiseen. Polarionin hyvin dokumentoitu SOAP-rajapinta mahdollistaa yleisimpien projekteihin ja tallennettuihin tietoihin liittyvien toimintojen käyttämisen. Polarionin SOAP-rajapinnan käyttöön kehitettiin Python-luokka, jonka tarkoituksena oli toimia ”tulkkina” Polarionin ja muiden mahdollisten järjestelmien välissä (ks. kuvio 23). *PolarionWebService*-luokka kehitettiin Pythonilla, koska ohjelmiston testauksessa käytettävässä HIL-ympäristössä voidaan suorittaa Python-skriptejä automatisoitujen testien yhteydessä.



KUVIO 23. Polarion SOAP ⇔ PolarionWebService.py ⇔ HIL

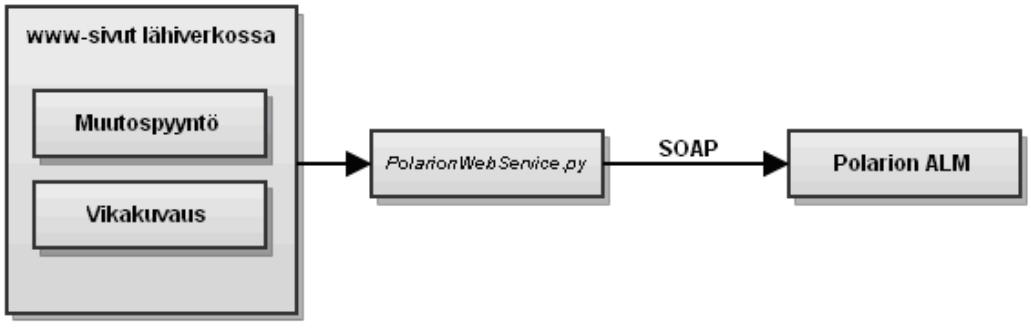
Kehitetyn Python-luokan tarkoituksena oli mahdollistaa automatisoitujen HIL-testien tulosten päivittäminen Polarionissa määriteltyihin testitapauksiin (ks. kuvio 24).

Testituloksen lisäksi Polarionin testitapaukseen lisättiin testatun ohjelmiston versio kommenttina. Tällä ratkaisulla saavutettiin jäljitettävyys automatisoiduista testeistä Polarioniin määriteltyihin testitapauksiin ja sitä kautta ohjelmiston vaatimuksiin. Liitteissä 1 ja 2 on otokset *PolarionWebService.py*-luokan toteutuksesta ja sen käytöstä HIL-ympäristössä.



KUVIO 24. Tiedonsiirto HIL-ympäristön ja Polarionin välillä

Kehitettyä Python-luokkaa voitiin käyttää myös muissa ympäristöissä. Useilta eri osapuolilta saapuvien kehitys- ja muutospyyntöjen sekä havaittujen vikojen keskitettyyn kirjaamiseen kehitettiin yksinkertaiset www-sivut (ks. kuvio 25). Sivujen tarkoitus oli havainnollistaa kehitetyn Python-luokan monikäyttöisyyttä ja mahdollista käyttötapausta. Tiedonsiirtoa www-sivujen ja Polarionin välillä testattiin käytännössä, mutta sitä ei käytetty todellisten kehitys- ja muutospyyntöjen tai vikojen kirjaamiseen.



KUVIO 25. Tiedonsiirto www-sivujen ja Polarionin välillä

5 POHDINTA

Ohjelmiston elinkaarenhallinta on aiheena hyvin laaja, koska se käsittelee ohjelmistokehitystä ja ohjelmiston elinkaarta kokonaisuutena keskittymättä varsinaisesti yksittäisiin vaiheisiin. Kirjallisuudessa ALM on koettu konseptina vielä kohtuullisen uudeksi ja siitä on kirjoitettu eri näkökulmista. Aihealue käsitti itsessään paljon tutkittuja aiheita, kuten vaatimustenhallinta tai konfiguraationhallinta. Yhdeksi ongelmaksi muodostui aiheen rajaaminen ja käsittely vain opinnäytetyön kannalta tärkeiden aiheiden osalta. Tämä aiheutti osaltaan haasteita käytännön kehittämistyössä, mutta yhteisiäkin tekijöitä kirjallisuudesta löytyi (*mm. jäljitettävyys, raportointi ja työkalujen integraatiot*) ja niitä pyrittiin käyttämään kehittämistyön perustana.

Polarionin käyttöönotolla saavutettiin ohjelmiston elinkaarenhallinnan kannalta tärkeää jäljitettävyyttä ja läpinäkyvyyttä Valtran ohjelmistokehitykseen sekä kykyä suunnitella ja seurata edistymistä tehokkaammin. Tietotyyppien ja työnkulkujen määrittely auttoi dokumentoimaan ja käsittelemään elinkaaren tuotoksia ja tapahtumia järjestelmällisellä tavalla. Polarionin avulla mm. ohjelmiston vaatimukset, toteutus ja testaus ovat jäljitettävissä ja läheisemmin yhteydessä toisiinsa.

Osa Polarionin käyttöönotolla saavutetuista hyödyistä oli vaikea arvioida jo opinnäytetyön aikana, koska Valtralla kehitettävän ohjelmiston elinkaari on usein suhteellisen pitkä (vuosia). Näin ollen myös käyttöönotolla saavutetut hyödyt jakautuvat pitkälle aikavälille. Joitain opinnäytetyön aikana kehitettyjä ratkaisuja ei ehditty vielä käyttää käytännössä. Opinnäytetyön aikana tehty työ ja lopputulokset kuitenkin palvelivat mielestäni hyvin työn tarkoitusta, eli Polarion ALM –ohjelmiston käyttöönottoa ja mukauttamista. Tavoitteen mukaisia hyötyjä tunnistettiin jo opinnäytetyön aikana.

ALM-ohjelmiston tehokas hyödyntäminen edellytti tutustumista useisiin eri ohjelmistokehityksen vaiheisiin sekä kykyä hahmottaa laajoja kokonaisuuksia. Tämä oli osaltaan haasteellista, mutta ammattitaidon kehittymisen kannalta hyvin tärkeää. Tärkeässä osassa oli soveltaa teoriassa käsiteltyjä menetelmiä Valtran ympäristön erityis-

piirteisiin. Useat ohjelmistosuunnittelun menetelmät eivät ole käytännöllisiä Valtran ympäristössä sellaisena, kuin ne kirjallisuudessa on esitetty. Sain opinnäytetyön aikana tärkeää käytännön kokemusta ohjelmistokehityksen käytännön haasteista ja menetelmistä ympäristössä, jossa kehitettävä ohjelmisto toimii osana laajaa sulautettua järjestelmää ja mekaanista tuotetta eli traktoria.

LÄHTEET

About Polarion Software n.d. Viitattu 30.01.2012
http://www.polarion.com/company/index.php

Chappel, D. 2008. What is Application Lifecycle Management.
Viitattu 13.01.2012
http://www.microsoft.com/global/applicationplatform/en/us/RenderingAssets/Whitepapers/What%20is%20Application%20Lifecycle%20Management.pdf

Hull E., Jackson K. & Dick J. 2004. Requirements Engineering.

Kallio, J. 2008. Vaatimustenhallinta ja sen kehittäminen ohjelmiston elinkaaren näkökulmasta. http://urn.fi/URN:NBN:fi:jyu-200808175658

Kääriäinen, J. 2011. Towards an Application Lifecycle Management Framework.
Viitattu 05.03.2012.
http://www.vtt.fi/inf/pdf/publications/2011/P759.pdf

Polarion Performance and Scalability. 2010. Viitattu 30.01.2012.
http://blog.polarion.com/archives/839

Rizzo, S. 2011. Agile, Requirements Management and Regulatory Compliance – A Practical Live Approach.
http://www.polarion.com/user/direct_register.php?dl=Agile-Requirements-Management-and-Regulatory-Compliance-A_Practical-Live-Approach.pdf

Schwaber, C. 2006. The Changing Face of Application Life-Cycle Management.
Viitattu 13.01.2012.
http://i.bnet.com/logos/whitepapers/Serena_Life_Cycle_Management.pdf

Shaw, K. 2007. Application Lifecycle Management for the Enterprise.
Viitattu 06.03.2012.
http://www.serena.com/docs/repository/company/serena_alm_2.0_for_t.pdf

SOAP Introduction n.d. Viitattu 16.03.2012.
http://www.w3schools.com/SOAP/soap_intro.asp

Tietoa Valtrasta n.d. Viitattu 05.01.2012.
http://www.valtra.fi/company/128.asp

LIITTEET

Liite 1. Otos PolarionWebService.py –luokan toteutuksesta

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
import os
import sys
import re

import logging
import suds
import csv

from suds.client import Client
from suds.sax.element import Element
from suds.sax.attribute import Attribute

from pprint import pformat

"""
Class: PolarionWebService()
Functionality for Polarion WebServices
"""
class PolarionWebService:
    def __init__(self):
        """
        Initialize SOAP Client.
        """
        # Logging config
        logging.basicConfig(level=logging.INFO)
        logging.getLogger('suds.client').setLevel(logging.CRITICAL)
        #logging.getLogger('suds.client').setLevel(logging.DEBUG)

        # Init SOAP Clients

        # Server prefix
        prefix = "http://agfinaps152.nordic.agcocorp.com" # Valtra
        # prefix = "http://alm.mod.agcocorp.com" # AGCO test server

        global builder
        builder = Client(prefix+'/polarion/ws/services/BuilderWebService?wsdl')

        global security
        security = Client(prefix+'/polarion/ws/services/SecurityWebService?wsdl')

        global project
        project = Client(prefix+'/polarion/ws/services/ProjectWebService?wsdl')

        global session
        session = Client(prefix+'/polarion/ws/services/SessionWebService?wsdl')

        global tracker
        tracker = Client(prefix+'/polarion/ws/services/TrackerWebService?wsdl')


    def login(self, username, password):
        """
        Login to Polarion WebService and set SOAP session headers
```

```
        """
        try:
            # Login
            result = session.service.logIn(str(username),str(password))

            # Parse sessionID from response envelope
            login_response = session.last_received()
            session_id = log-
in_response.getChild("soapenv:Envelope").getChild("soapenv:Header").getChild("ns1:
sessionID").getText()
            session_header = Element('sessionID',
ns=('ns1','http://ws.polarion.com/session')).setText(session_id)
            session_header.append(Attribute("SOAP-ENV:mustUnderstand", "0"))
            session_header.append(Attribute("SOAP-ENV:actor",
"http://schemas.xmlsoap.org/soap/actor/next"))

            # Set SOAP headers
            builder.set_options(soapheaders=session_header)
            security.set_options(soapheaders=session_header)
            project.set_options(soapheaders=session_header)
            session.set_options(soapheaders=session_header)
            tracker.set_options(soapheaders=session_header)


            print "Logged in Polarion WebService as ["+username+":"+password+"]"
            return True

        except suds.WebFault, f:
            print f
            return False
        except Exception, e:
            print e
            return False


    def updateWorkItemResolution(self, project_id, wi_id, resolution):
        """
        Updates Resolution of WorkItem
        """
        try:
            session.service.beginTransaction()

            # get WI
            wi = tracker.service.getWorkItemById(project_id, wi_id)

            # update only if the current resolution is different from the new
resolution
            if(wi.resolution.id != resolution):
                wi.resolution.id = resolution
                tracker.service.updateWorkItem(wi)
            else:
                pass

            print "Resolution of ["+wi_id+"] updated to ["+resolution+"]"

        except suds.WebFault, f:
            print f
        except Exception, e:
            print e
        finally:
            session.service.endTransaction(False)


    def createComment(self, project_id, wi_id, comment):
```

```
"""
Create new comment to WorkItem
"""
try:
    session.service.beginTransaction()

    # get WI
    wi = tracker.service.getWorkItemById(project_id, wi_id)

    # create comment
    c = tracker.factory.create("ns4:Text")
    c.type = 'text/html'
    c.content = str(comment)
    c.contentLossy = False
    tracker.service.createComment(wi._uri, c)

    print "Comment ["+comment+"] added to ["+wi_id+"]"

except suds.WebFault, f:
    print f
except Exception, e:
    print e
finally:
    session.service.endTransaction(False)
```

Liite 2. Esimerkki PolarionWebService.py –luokan käytöstä HIL testissä.

```
# PolarionWebSerivice class example usage from Automation Desk HIL test sequence
ws = PolarionWebService()
ws.login('HIL','P4ssW0rd')

# Get data related to test case from local AutomationDesk variables
test_id = _AD_.TestStepVariables.Id
test_result = _AD_.TestStepVariables.TestResult
sw_ver = _AD_.TestStepVariables.SwVersion

# Update Polarion Work Item with test result and add tested software version as a
comment
ws.updateWorkItemResolution('hiekkalaatikko', test_id, test_result)
ws.createComment('hiekkalaatikko', test_id, sw_ver)
```