EVTEK-ammattikorkeakoulu
Mediatekniikan koulutusohjelma

Sara Kapli

Testausprosessimalli, yksikkötestaus ja testausympäristön automatisointi Java-ympäristössä

Insinöörityö 10.5.2005

Työn ohjaaja: Production Director
Johannes Verwÿnen
Työn valvoja: yliopettaja Harri Airaksinen

EVTEK-
AMMATTIKORKEAKOULU

INSINÖÖRITYÖN
TIIVISTELMÄ

Tekijä Otsikko Sivumäärä Aika	Sara Kapli Testausprosessimalli, yksikkötestaus ja testausympäristön automatisointi Java-ympäristössä 65 10.5.2005
Koulutusohjelma	mediatekniikka
Ohjaaja Valvoja	Production Director Johannes Verwÿnen yliopettaja Harri Airaksinen

Insinöörityön tavoite oli kehittää ohjelmistoyritykselle Java-kehitykseen sopiva testausprosessi soveltaen yrityksen tilanteen kannalta parhaita puolia kahdesta erilaisesta metodologiasta: perinteisestä vesiputousmallisesta ohjelmistokehityksen prosessimallista ja ketterästä menetelmästä nimeltä Extreme Programming.

Kehitetyssä hybriditestausmallissa päädyttiin Extreme Programming -painotteiseen tapaan. Siitä ovat peräisin testauslähtöinen kehittäminen, tiivis yhteistyö ja vapaamuotoinen viestintä ryhmässä. Perinteisestä ohjelmistokehityksen mallista otettiin tavat tehdä dokumentaatioita ja ottaa erillinen testaaja projektiryhmään.

Tavoitteena oli myös luoda toimivat käytännöt ohjelmistojen yksikkötestaukseen, jonka apuvälineenä käytettiin automatisoiduille yksikkötesteille suunniteltua JUnit-kehystä.

Yksityiskohtaisen yksikkötestauksen vaatima testausympäristön automatisointi oli eräs insinöörityön tavoitteista. Testausympäristö automatisoitiin Ant-koostustyökalun ja Maven-hallintaohjelman avulla.

Lopputuloksena yritykselle syntyi sen tarpeisiin räätälöity kevyt ja muuntuva hybridi testausprosessimalleista. Yksikkötestaukseen löydettiin käytännöt, jotka havaittiin hyviksi. Niitä käytettiin tulevan projektin suunnitelmassa. Yksikkötestausympäristö automatisoitiin, joten tulevissa projekteissa testaus sujuu vielä sutjakkaammin. Kertaalleen suoritetun automatisoinnin pohjalta seuraavien projektien testien automatisointiin on pienempi kynnys. Testausprosessimalli kehitettiin yritykselle jatkuvaan käyttöön, joten sitä sovelletaan yrityksessä myöhemmissä Java-projekteissa.

Hakusanat	Extreme Programming, yksikkötestaus, JUnit, TDD, automatisoitu testaus, Maven, testauslähtöinen kehitys

ESPOO–VANTAA INSTITUTE OF TECHNOLOGY

ABSTRACT

Author	Sara Kapli
Name of Thesis	Test process model, unit testing and automating test environment in Java development environment
Pages	65
Date	10 May 2005
Degree Programme	Media Technology
Instructor	Johannes Verwÿnen, Production Director
Supervisor	Harri Airaksinen, Principal Lecturer

The main goal of the thesis was to design a test process model suitable for Java development. Two different methodologies were examined for creating the test process for the software company. Most suitable parts of both, the traditional waterfall life cycle model of software development and an agile method, Extreme Programming, were used to tailor the process to fit to the company's current situation.

Extreme Programming became the emphasized methodology in the designed hybrid process model. Close co-operation and informal communication within a project team along with test driven development were picked out from Extreme Programming. Style of documentation and having a separate tester in the team were adopted from the traditional software development model.

Conventions and practices were created as a part of the thesis for unit testing the software. The unit test framework used was JUnit. For the designed process model to be implementable and fully work, automated testing environment is crucial. Test environment was automated using Maven, a software project management and comprehension tool, and Ant, a build tool.

As a result, the company got a lightweight and flexible hybrid test process model tailored for their needs. Conventions and practices that were created for the unit tests were found good and those were used in a plan for an upcoming project. Unit test environment was automated, and experience from that helps automating the upcoming project. The test process model was developed for the company for continuous use and the model will be applied to the company's future Java projects.

Keywords	Extreme Programming, unit testing, automated testing, JUnit, Maven, TDD, test driven development

Sisällys

1 Johdanto

Monessa pienessä yrityksessä ovat usein sekä aika että resurssit vähissä. Ohjelmistotuotannossa se tarkoittaa tiukkaa aikataulua pienellä tekijämäärällä. Varaa virheiden korjaamiseen ei juuri jätetä, eikä koskaan ole ainakaan aikaa tehdä kaikkea uusiksi. Virheiden määrää ja ohjelmistokehityksen riskejä voidaan pienentää ottamalla testaaminen mukaan mahdollisimman varhaisessa vaiheessa. Insinöörityön tarkoituksena on kehittää Fathammer Oy:lle toimiva Java-kehitykseen soveltuva testauskäytäntö sekä lisäksi testaussuunnitelma pelipalvelinohjelmiston kehitykseen pienemmän projektinhallintaohjelmiston testauksesta saatujen kokemusten avulla.

Fathammer Oy perustettiin lokakuussa vuonna 2000. Aluksi yritys teki laitteistosta riippumatonta 3D-pelialustaa mobiililaitteisiin ja siihen pari peliä teknologianäytteeksi. Nyt strategia on siirtymässä väliohjelmiston lisensoimisesta mobiilipelien julkaisijaksi ja tuottajaksi. Oman pelituotannon lisäksi joukko ulkopuolisia yhteistyöyrityksiä tekee pelejä Fathammerin julkaistavaksi. Fathammer on keskisuuri yritys, joka työllistää noin 40 henkilöä. Työntekijöistä muutama on Etelä-Koreassa sijaitsevassa tytäryhtiössä.

Markkinoiden tämänhetkinen tilanne suosii pelejä, jotka on toteutettu Java-teknologialla. Puhelinoperaattorit ovat juuri nyt suurin mobiilipelien jakelija. Operaattorit suhtautuvat nihkeästi laitekohtaisesti toteutettuihin peleihin eivätkä halua ottaa niitä listoilleen. He pitävät laitekohtaisia pelejä turvallisuusuhkina, sillä niiden toimintaa laitteessa ei ole rajoitettu (*sandboxed*) samalla tavoin kuin Java-pelien. Tämä on pakottanut Fathammerin harkitsemaan vakavasti Java-kehityksen aloittamista peliprojekteissa. Java on yleisesti todettu hyväksi valinnaksi tässä työssä käsiteltävien X-Forge Onlinen ja Piiskan kaltaisiin palvelinpohjaisiin ohjelmistoihin. Näistä syistä yrityksessä aletaan pikku hiljaa kehittää ohjelmistoja C++:n lisäksi Javalla. Fathammerilla ei ennestään ole Java-kehitysprosessia eikä myöskään Java-kehitykseen sopivaa testauskäytäntöä. Testausprosessi ja -käytännöt rakennetaan Piiskan

testaustyön ohella, ja siitä hankittujen kokemusten avulla käytäntöjä sovelletaan aluksi X-Forge Onlinen testaussuunnitelman kirjoittamisessa.

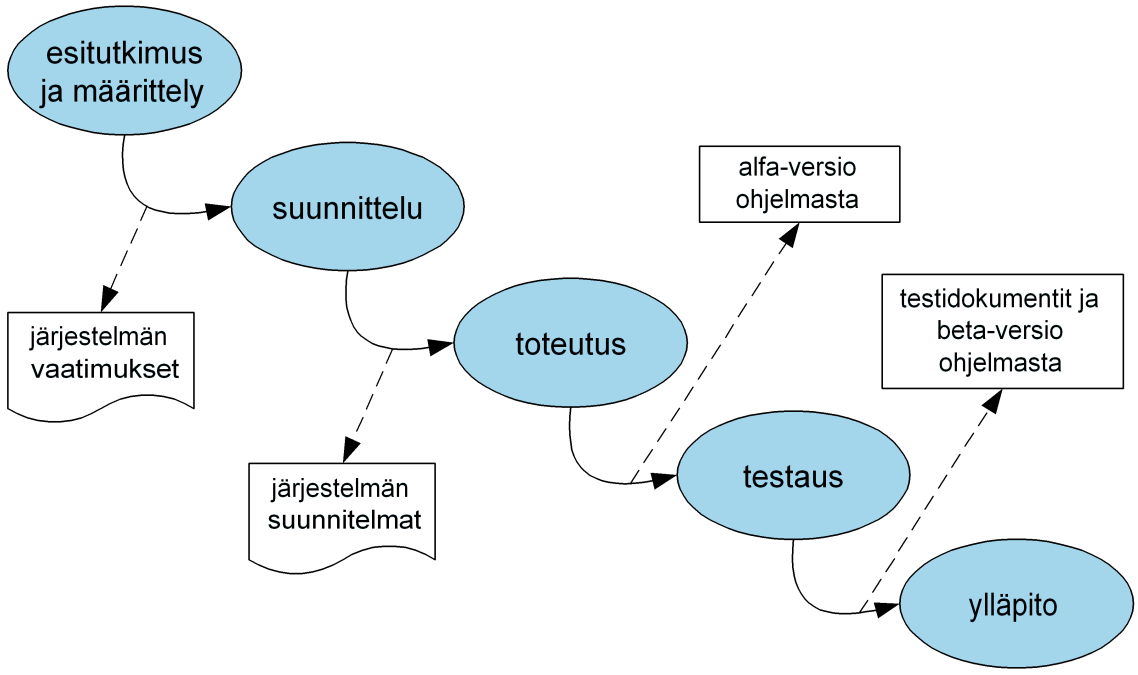
2 Ohjelmistotuotannon prosessimallit

2.1 Tarkasteltavat prosessimallit

Fathammer Oy:hyn ei ollut muotoutunut C++-kehityksen yhteydessä kunnollisia testauskäytäntöjä, joita olisi voinut mukauttaa Java-testauksen prosessien pohjaksi, vaan prosessimallin suunnitteluun voitiin lähteä lähes puhtaalta pöydältä, tietenkin yrityskulttuuri huomioon ottaen. Suuntaviivoja yrityksen testausprosessille haettiin sekä perinteisestä ohjelmistotuotannosta että erilaisista ketteristä menetelmistä (*agile methods*). Ketteristä menetelmistä tutkittiin lähemmin erittäin vapaamuotoista Extreme Programming -menetelmää (*XP*). Osana insinöörityötä näistä menetelmistä koottiin omaksi testauksen prosessimalliksi yritykselle sen tilanteeseen parhaiten sopivat käytännöt.

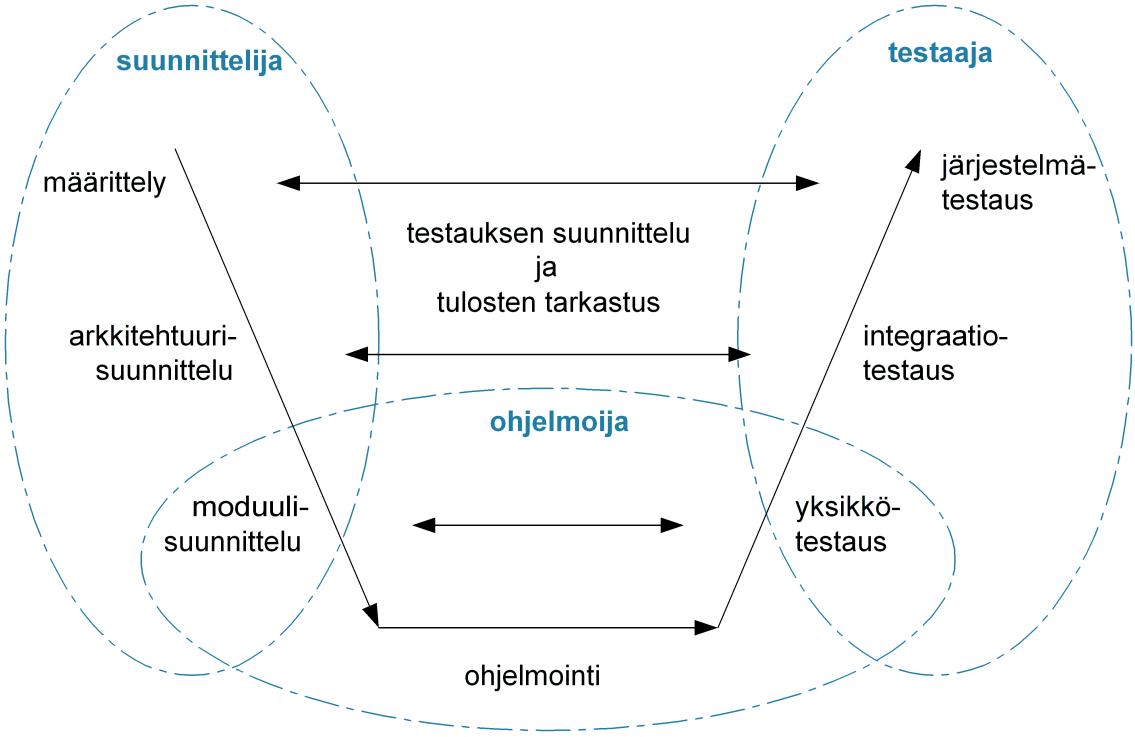
2.2 Perinteinen ohjelmistokehitys

Perinteisessä ohjelmistotuotannossa kehityksen elinkaarimallina käytetään tavallisesti vesiputousmallia, joka näkyy kuvassa 1 [Rie96].



Kuva 1. Vesiputousmallinen elinkaari [Rie96]

Vesiputousmallinen prosessi etenee lineaarisesti vaiheittain, ja testaus on oma erillinen vaiheensa elinkaaren loppupäässä. Perinteinen tapa testata on yleensä niin sanotun V-mallin mukainen. Kuvassa 2 esitetyssä V-mallissa testaus on jaettu kolmeen eri tasoon [JoE94]. V-mallin ydinajatus on, että testit suunnitellaan testaustasoa vastaavalla suunnittelutasolla ja testauksen tuloksia vertaillaan aina saman tason toteutuksen määrittelyssä tehtyihin dokumentteihin [HaM02, s. 286]. Täten siis määrittelyvaiheessa suunnitellaan järjestelmätason testaukset, integraatiotestaus samalla kuin arkkitehtuurisuunnittelu ja moduulien suunnittelun yhteydessä suunnitellaan myös yksikkötestaukset.



Kuva 2. Testauksen V-malli ja kehittäjien vastuualueet [HaM02, s. 286—288]

Moduulin kehittäjä itse toteuttaa ja suorittaa yksikkötestauksen toteuttamalleen moduulille. Yksikkötestaus on tämän takia luonteeltaan lasilaatikkotyyppistä (*white box testing*) testausta, sillä kehittäjä tuntee toki oman ohjelmansa toteutuksen hyvin yksityiskohtaisesti. Integraatiotestaus etenee yleensä

jotakuinkin yhtäaikaisesti yksikkötestauksen kanssa. Järjestelmätasolla testauksen tekijän täytyy olla ohjelmiston kehityksestä riippumaton, sillä järjestelmätestaukset ovat mustalaatikkotyyppisiä (*black box testing*) eli tarkoitettu suoritettavaksi tuntematta ohjelman toteutusta. [HaM02, s. 286—289] Kuvaan 2 on merkitty testaustasojen lisäksi myös eri kehittäjien roolien vastuunjako.

Perinteisen ajattelun mukaan testaus on huomattavan kallista ja saattaa muodostaa ohjelmistoprojektin kustannuksista valtaosan. Mitä korkeammalla tasolla – eli myöhempään – V-mallisessa testauksessa virheet löydetään, sitä kalliimpaa virheiden korjaaminen on. Etenkin regressiotestausta pidetään kalliina, sillä testien automatisointi ei ole perinteisessä ohjelmistokehityksessä itsestään selvää ja aikaavievät testit joudutaan ajamaan käsin uudestaan heti, kun yksikin virhe ohjelmistossa korjataan. [HaM02, s. 288]

Yksikkötestaus tehdään perinteisesti vasta moduulin valmistumisen jälkeen [HaM02, s. 287]. Testauksen tekee samaisen yksikön kehittäjä. Kun moduulin tekijä ja testaaja ovat sama henkilö, yksikkötestit alkavat helposti heijastaa ohjelman senhetkistä toiminnallisuutta, joka ei ole välttämättä haluttu, saati oikea, toiminnallisuus. Vaarana on, että kehittäjä tulee sokeaksi omille virheilleen, eikä tahattomasti väärin rakennettu yksikkötesti paljasta kaikkia ongelmia. Tällöin virheet ja ongelmakohdat löydetään vasta testauksen korkeammilla tasoilla ja niiden korjaaminen tulee aina vain kalliimmaksi. Pahimmassa tapauksessa ohjelman kuntoon saaminen tulee liian työlääksi ja on pakko aloittaa alusta.

Perinteinen integraatiotestaus etenee ohjelmistorakennepuuta pitkin joko kokoavasti tai jäsentävästi. Tällä tavoin suoritettavassa integraatiotestauksessa ohjelmiston rakenne asetetaan toiminnallisuuden edelle. Tämä saattaa olio-keskeisessä toteutuksessa johtaa siihen, että ohjelmassa on rakenteellisesti mahdollisia, mutta toiminnallisesti mahdottomia suoritusreittejä. [JoE94]

Perinteisessä ohjelmistokehityksessä painopiste on ohjelmiston perinpohjaisella määrittelyllä, suunnittelulla ja toteutumisen dokumentoinnilla enemmän kuin itse ohjelman tekemisellä. Perinteistä ohjelmistokehitystä kutsutaan kirjallisuudessa myös suunnitelmalähtöiseksi ohjelmistokehitykseksi (*plan- driven development*). Erilaiset dokumentit ovat pääasiallinen viestintätapa eri projektien, alaisten ja johtajien sekä saman organisaation kehittäjien välillä. Kehitysprosessit yleisesti ovat jäykkiä ja muodollisia. Suunnitelmalähtöinen kehitysprosessi on parhaimmillaan silloin, kun

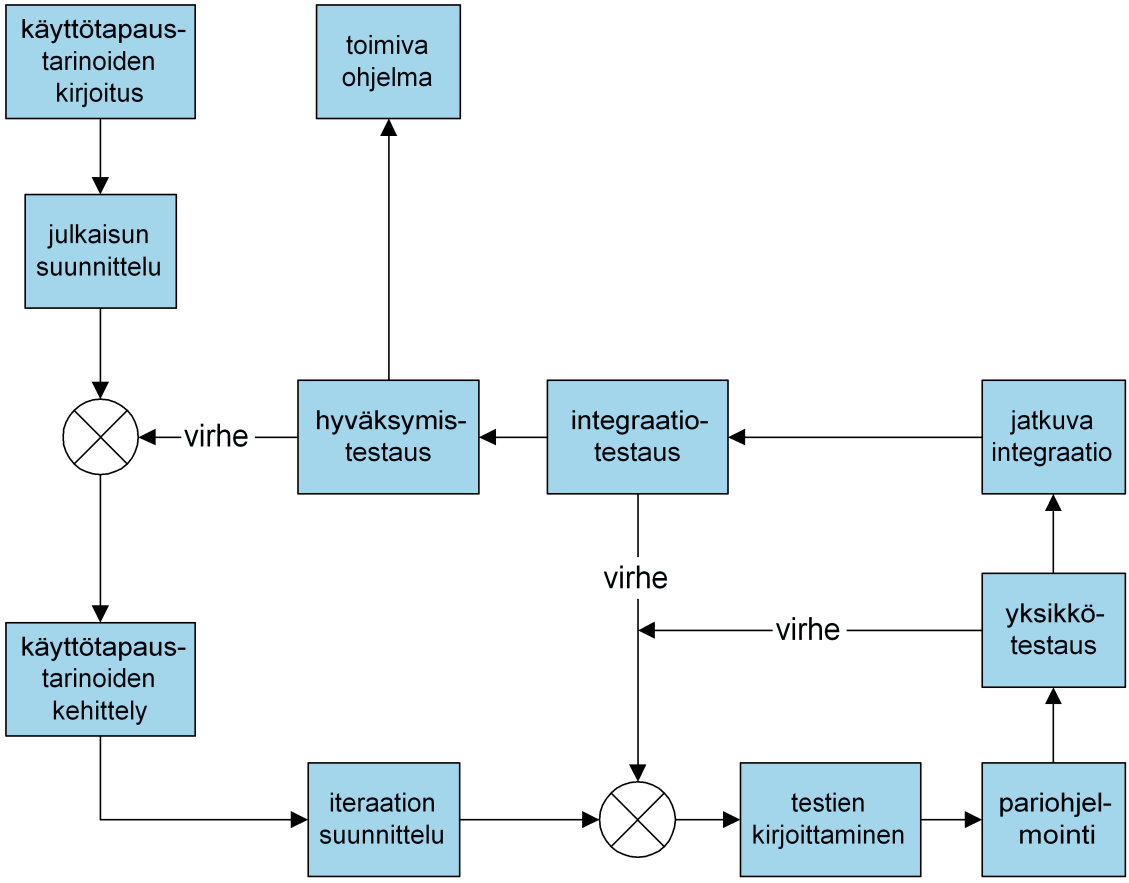
- kehitettävänä on kohtalaisen iso ja monimutkainen järjestelmä
- järjestelmä on turvatasoltaan kriittinen tai muusta syystä tarvitaan korkea luotettavuuskerroin
- järjestelmän määrittelyjä ei tarvitse muuttaa
- järjestelmä toimiii vakaassa ympäristössä [BoT03].

2.3 Extreme Programming -metodologia

Extreme Programming -metodologiassa (*XP*) ei ole selkeää elinkaarimallia, vaan sen tekemistä on suorastaan vältelty [McB02]. XP on inkrementaalinen ja iteratiivinen prosessimalli. Tarkoitus on, että toistetaan toimintoja riippumatta kehitystyön vaiheesta tai aikajänteestä ja näin luodaan kauttaaltaan samankaltainen prosessi. XP-tyylisen projektin kulku ja sen iteraatiosilmukat näkyvät kuvassa 3. Menetelmän prosessikulusta voidaan erottaa muutamia päävaiheita, joilla on joitakin yhteneväisyyksiä ja joitakin eroavaisuuksia perinteisen vesiputousmallin kanssa. Päävaiheet ovat

- tutkimus (*exploration*)
- suunnittelu (*planning*)
- ensimmäiseen julkaisuun tähtäävät iteraatiot (*iterations to first release*)
- tuotannollistaminen (*productionizing*)
- ylläpito (*maintenance*)
- projektin kuolema (*death of project*) [McB02].

Suurin eroavaisuus perinteiseen vesiputousmalliin verrattuna on erillisen testivaiheen täydellinen puuttuminen. XP-menelmissä testaaminen otetaankin alusta asti osaksi kehitysprosessia.



Kuva 3. XP-projektin vuokaavio iteraatiosilmukoineen [Ans03]

Ohjelmaosan kehittäjä kirjoittaa yksikkötestit. Yksikkötestaus ja ohjelmointi tehdään yhtäaikaisesti. Se on perinteistä ohjelmointia nopeampaa ja antaa kehittäjälle välitöntä palautetta sekä luottamusta omaan ohjelmaan [Bec00]. Useimmiten XP-tyylisessä ohjelmistoprojektissa käytetään testauslähtöistä kehitystä (*test-driven development, TDD*), jossa ohjelmapätkälle rakennetaan yksikkötesti, ennen kuin riviäkään itse testattavasta yksiköstä on kirjoitettu [BuC03; Bai03]. Testien tekemiseen ja ohjelmoimiseen tulee TDD:ssä muutaman vaiheen toistumisesta syntyvä rytmi. Nämä vaiheet ovat seuraavat:

1. Kirjoita testi.

2. Käännä testi.
3. Aja testi, joka ei mene läpi. Aluksi se johtuu siitä, että itse testattava ohjelma puuttuu.
4. Kirjoita varsinainen, testattava, ohjelma.
5. Käännä ohjelma.
6. Aja testi, joka onnistuu. Aikaisempia vaiheita 3—5 toistetaan, kunnes testi menee läpi.
7. Poista toisto ja uudelleenjärjestele lähdekoodi.

Vaiheita toistetaan, kunnes varsinainen ohjelma on hyvin jäsennettyä, tarkoituksenmukaista ja kaunista ja kaikki tehdyt testit onnistuvat eikä ohjelman osasta keksi enempää testattavaa. [Bec02]

Yksikkötestien onnistumisprosentin täytyy olla 100, ennen kuin testausta jatketaan integraatiovaiheeseen [BuC03; Bec02]. Integraatiotestaus on jatkuvaa siten, että ohjelman osa liitetään muuhun ohjelmaan heti, kun osalle tehtävät yksikkötestit onnistuvat [AnS03]. Integraatiotestit koko ohjelmalle ajetaan ainakin kerran päivässä, mieluiten useammin. Kun integraatio tehdään usein, virheet huomataan, ennen kuin niistä tulee osa ohjelman rakenteita.

XP-menetelmiin kuuluu yhtenä osana pariohjelmointi. Ohjelmistoa kehitetään parityöskentelynä niin, että kaksi kehittäjää istuu saman tietokoneen ääressä ja työskentelee saman ohjelmaosan kanssa [Bec00]. Parityöskentelyn ansiosta sokeus omille virheille vähenee, kun samaa ongelmaa ratkovat kahdet aivot. Toinen voi huomauttaa toiselle, jos tämä on tekemässä jotain väärin. Testauslähtöisessä kehityksessä pari aloittaa ohjelmoimalla yksikkötestit tehtäväksi annetulle ohjelman osalle. Kun on todettu, että juuri kirjoitettu testi ei mene läpi, pari käy tehtävän ohjelmoinnin kimppuun. Pari suorittaa yhdessä tekemälleen ohjelmalle sekä yksikkö- että integraatiotestit. Erilliset testaajat tai testiryhmät puuttuvat kokonaan XP-menetelmistä.

Testauslähtöisen kehityksen, pariohjelmoinnin ja jatkuvan integraation ansiosta virheet ja niiden vaikutusalue huomataan mahdollisimman aikaisin, joten

virheet voidaan korjata nopeasti. Pikainen korjaaminen on taloudellisesti tehokasta. Usein ja eri tasoilla tapahtuvan kurinalaisen testauksen jälkeen voidaan luottaa tuotetun ohjelman laatuun. Jotta tällainen kattava ja nopeatahtinen testaaminen olisi mahdollista ja järkevää, vaaditaan testiympäristöltä laajaa automatisointia. Testiympäristön automatisointi on olennainen osa Extreme Programming -metodologiaa, ja monet menetelmistä pohjautuvat siihen [BuC03].

XP-menetelmien viestintä perustuu siihen, että niitä käyttävän ryhmän jäsenet istuvat lähellä toisiaan, mieluiten samassa huoneessa. Suurin osa viestinnästä on suusanallista ja vapaamuotoista [Bec02]. Dokumentteja tehdään hyvin vähän. Prosessissa käytettyjä kirjallisia tuotoksia ovat lähinnä asiakkaan kanssa tehdyt käyttötapaustarinat (*user story*), joiden pohjalta ohjelmaan rakennetaan toiminnallisuuksia [AnS03]. Painopiste on itse ohjelman tekemisessä, sillä kurinalaisesti testattu ja toteutettu ohjelma testeineen on tarvittava dokumentaatio itsessään [Jef04]. Perinteisiä suunnittelu- ja toteutusdokumentteja ei ole, sillä niiden tekemistä ja ylläpitoa pidetään ylimääräisenä työtaakkana.

XP-tyylinen prosessi on hyvin vapaamuotoinen ja muuntuva, vaikkakin kurinalainen. XP:n menetelmät ovat parhaimmillaan silloin, kun

- kehitysryhmät ovat pieniä
- sekä asiakas että loppukäyttäjät sijaitsevat lähellä ja ovat valmiita sitoutumaan projektiin
- kehitettävät järjestelmät ovat pieniä tai järjestelmien määrittelyt tai ympäristö ovat ailahtelevaisia [BoT03].

2.4 Yrityksen malli

Ohjelmistojen kehittäminen on Fathammer Oy:ssä hyvin kiivastahtista, ja projektien tavoitteet sekä määrittelyt muuttuvat vähän väliä. Vesiputousmallinen elinkaari ei ole yrityksen tarpeisiin riittävän joustava ja mukautuva [Ver05; Ham95]. Prosessimallia testaukseen luodaan Java-kehitystä

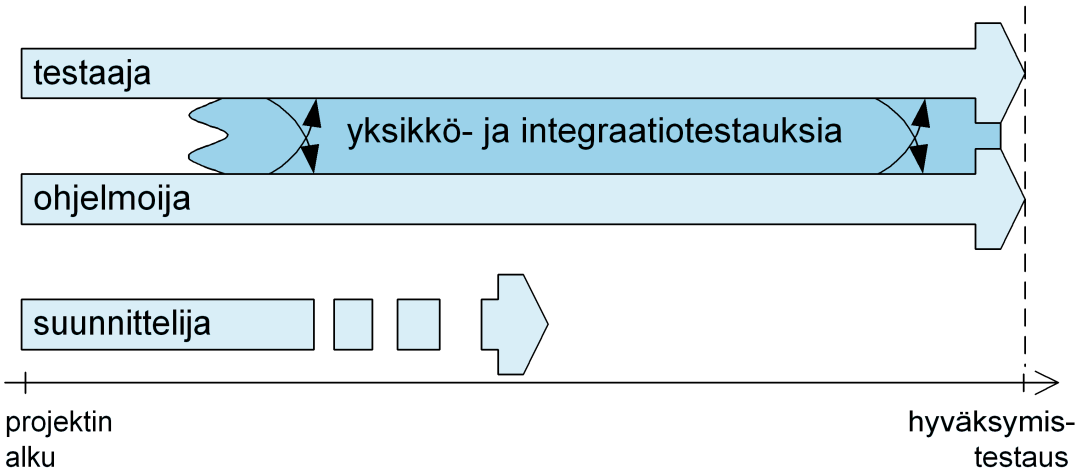
varten, ja olio-ohjelmointiprosessi on harvoin vesiputousmallin mukainen [JoE94]. Vapaamuotoinen prosessimalli, joka on lähempänä inkrementaalisia malleja, vastaa paremmin yrityksen tarpeita. Ohjelmistokehityksen riskejä pyritään hallitsemaan ottamalla testaus mukaan kehitykseen mahdollisimman varhaisessa vaiheessa [Ver05].

Kiivastahtisessa kehityksessä ja tiukoilla aikatauluilla aikaa ei kannata tuhlata ylenpalttisesti dokumentaatioiden tekemiseen. Tähän menessä Fathammerin aiemmassa ohjelmistotuotannossa on kuitenkin ollut ongelmana dokumentaatioiden puute. Java-kehitykseen haluttiin siksi tuoda heti alussa tapa tehdä ja ylläpitää muutamaa tärkeintä dokumenttia. Ne ovat testaussuunnitelma, projektisuunnitelma ja järjestelmän tekninen kuvaus. Testaussuunnitelma ja projektisuunnitelma kirjoitetaan IEEE:n standardeja seuraten. Käytetyt standardit ovat 829-1998 testausdokumentaatiolle ja 1058-1998 ohjelmistoprojektin hallinnointisuunnitelmalle [IES98a, IES98b]. Tekninen kuvaus on vapaamuotoinen dokumentti, jolla pyritään helpottamaan järjestelmän arkkitehtuurin hahmottamista. Dokumenteista ei ole tarkoitus tehdä kaikenkattavia järkäleitä, vaan suhteellisen keveitä ja ajantasaisia kuvauksia projekteista. Projekteista tahdotaan pysyviä dokumentteja, mutta niiden kirjoittamisella ei haluta kuormittaa kehittäjiä turhaan. [Ver05]

Yrityksellä ei ole resursseja parityöskentelyyn [Ver05]. Ohjelmoijien sokeutta omille virheilleen ehkäistään ottamalla Java-projekteihin erillinen testaaja. Eri henkilö testaajana on hyvä myös siksi, että ohjelmoija ei välttämättä tiedä tarpeeksi testaamisesta [McB02]. Erillisellä testaajalla voidaan siirtää työtaakkaa ohjelmoijalta pois, samalla kun testaaja tuo omaa ammattitaitoaan projektiin. Testaaja on tiiviisti osa projektiryhmää. Viestintä ryhmän sisällä hoituu XP-tyyppisesti suullisesti ja tussitaululla, kun projektiryhmän jäsenet istuvat kuitenkin lähekkäin samassa toimistossa. [Bur03; CrH03]

Yksikkötestit pyritään tekemään testauslähtöisesti ennen varsinaista ohjelmaa tai viimeistään samanaikaisesti ohjelman kirjoittamisen kanssa [Ver05]. Kun

ryhmässä on erillinen testaaja, tämä edellyttää rajapintojen suunnittelua etukäteen. Toinen edellytys on, että testaaja on mukana projektissa alusta asti, myös ohjelman suunnittelussa [Mar03]. Niinpä testaaja otetaan projektiin heti alussa, kun projektisuunnitelma, testaussuunnitelma ja tekninen kuvaus tehdään. Olio-pohjaisista ohjelmista ei voi tehdä perinteisentyyppistä selvää integraatiopuuta (*decomposition tree*), joten integraatiotestaus etenee XP-tyyppisesti jatkumona [JoE94]. Yksikkö voidaan integroida heti, kun sen yksikkötestit menevät läpi. Kuvassa 4 havainnollistetaan projektiryhmän jäsenten rooleja ja aikataulutusta sekä testaajan ja ohjelmoijan tiivistä yhteistyötä koko ohjelman kehityksen ajan. Kuvassa näkyvän kaavion suunnittelija ja ohjelmoija voivat olla sama henkilö.



Kuva 4. Yrityksen prosessimalli

Vaikka yksikkötestit mielellään tehdään ennen ohjelmaa, testauksen aikataulu halutaan pitää riippumattomana itse ohjelman kehityksen vaiheesta. Yksikkötestit tulevat olemaan harmaalaatikkotyyppisiä. Testit tehdään pääasiassa dokumentaation mukaisesti. Testaajan ei ole tarkoitus tuntea perinpohjaisesti testattavan ohjelman osan toimintaa. Toisaalta yksikkötestejä ei voi tehdä lasilaatikkotyyppisesti, koska testaajalla tuskin on pikkutarkkaa dokumentaatiota kaikesta. Näistä syistä rikkinäisten testien syy selvitetään testaajan ja ohjelmoijan yhteistyönä, vaikka tietenkin ohjelmoija on ensisijaisesti vastuussa testien läpimenosta [Mar03]. Projektin aikana

ohjelmoija voi käydä katsomassa automaattisen testausympäristön tuottamat testitulokset projektin www-sivuilta. Jos testejä rikkoontuu, ohjelmoija tarkistaa ensin oman lähdekoodinsa. Ellei ohjelmassa näytä olevan korjattavaa, on vuorossa keskustelu testaajan kanssa. Prosessi on helppo ja inhimillinen. Ongelmat hoituvat nopeasti, kun projektiryhmä työskentelee samassa tilassa [Becoo].

3 Testausympäristön automatisointi

3.1 Automatisoinnin edellytykset

Fathammer Oy:lle tehty testausprosessimalli nojaa vahvasti testauslähtöiseen kehitykseen. Testauslähtöinen kehitys vaatii toimiakseen pitkälle vietyä testauksen automatisointia. Muuten prosessista tulee liian raskas, eikä sen läpivientiin riitä resursseja [Bec00]. Insinöörityön yksi osa-alue oli toteuttaa yritykselle automatisoitu yksikkötestausympäristö. Yksikkötestien kehyksinä käytetään JUnit-kehystä, ja muu testausympäristö rakennettiin sen mukaan [Ver05].

Testausympäristön automatisointi yksikkötestausta varten vaatii useiden eri osa-alueiden hallintaa, joista jokaiselle on useimmiten oma erikoistunut sovelluksensa. Ennen testaamista ohjelman lähdekoodi on käännettävä Java-virtuaalikoneen hyväksymäksi tavukoodiksi. Usein vain muuttuneiden luokkien kääntäminen riittää. Esimerkiksi RMI-teknologiaa (*Remote Method Invocation*) käytettäessä kaikki luokat on aina käännettävä uusiksi. Koko projektin luokat on joka tapauksessa hyvä kääntää uudestaan ainakin öisin, kun kääntämiseen kuluva aika ei ole kehittäjien työajasta pois. Luokkien testaamista varten täytyy olla jonkinlainen kehys, joka huolehtii testien ajosta ja tuloksien kertomisesta erityisesti, kun jotain menee vikaan. Testaamisen jälkeen testitulokset täytyy saada ihmisen tulkittavaan muotoon ja kehittäjien saataville.

Automatisointiin on saatavilla useita hyötyohjelmia, joista piti valita sopivimmat yrityksen tarpeisiin. Näiden työkalujen valintaan osallistui myös yrityksen tekninen ylläpito, sillä se on vastuussa yrityksen tietoturvasta. Se ylläpitää kehittäjien käyttämiä järjestelmiä työkaluineen. Tekninen ylläpito lisäksi avustaa kehittäjiä ympäristöön liittyvissä ongelmissa.

3.2 Työkalut

Automatisointiin valittiin avoimen lähdekoodin ohjelmia [NLL05]. Ohjelmilla on tukenaan aktiiviset käyttäjä- ja kehittäjäyhteisöt, joten ohjelmien toiminta on tunnettua ja apua on ongelmatilanteissa saatavilla. Valitut avoimet ohjelmat tukevat toisiaan hyvin. Osa työkaluista oli jo yrityksessä käytössä, kuten versionhallintaohjelma CVS. CVS:ää on kehitetty riittävän kauan, että se on käytännössä todettu luotettavaksi ja vakaaksi ohjelmaksi. Käytettävyydeltään CVS ei ole parhaita, mutta siihen on saatavilla peruskäyttöä helpottavia apuohjelmia vaikkapa graafista käyttöliittymää halajavalle. CVS on saatavilla useille käyttöjärjestelmille, ja se on hyvin tuettu myös eri sovelluskehittimissä, kuten Eclipsessä.

Valittu Java-kääntäjä on Java-ohjelmointikielen kehittäjän Sun Microsystemsin oma kääntäjä. Suuren ja kokeneen yrityksen tekemänä se on luotettava valinta. Kääntäjä on saatavilla monelle käyttöjärjestelmälle. Samoista syistä käytetään ajoympäristönä Sunin Java-virtuaalikonetta. Kääntäjän optimointikyvyillä ei ole niinkään väliä insinöörityöhön kuuluvien kahden projektin ohjelmissa, sillä sekä Piiska että X-Forge Online tulevat toimimaan laitteistoltaan suorituskykyisissä palvelimissa. Ajoympäristön suorituskykyä ei vastaavista syistä pidetty huolenaiheena. Laitteistokapasiteetin lisääminen on yritykselle suhteellisen pieni investointi verrattuna työn hintaan. Toisaalta, jos ohjelmien ajaminen alkaa tuntua hitaalta tai muistin määrä loppua, voidaan sekä kääntäjää että ajoympäristöä helposti vaihtaa, ellei uuden laitteiston hankkiminen tule kysymykseen.

Ohjelmien koostamistyökaluna (*build tool*) käytetään Ant-ohjelmaa. Se on ohittanut suosiossa Unixin make-ohjelman, ja siitä on tullut lähes standardinomainen työkalu Java-kehittäjille. Itsekin Java-pohjaisena Ant on käytännössä käyttöjärjestelmästä riippumaton. Ant toimii kokeillusti useammalla eri käyttöjärjestelmällä. [Ant05]

Testien kehyksinä käytetään toistuvia Java-kielisiä yksikkötestejä varten tehtyä JUnit-kehystä. JUnit on Java-kielellä toteutettu ja lähdekoodiltaan avoin. Unit-testikehyksiä on myös muilla kielillä toteutettuja, ja ne on tarkoitettu testaamaan toteuskielellään kirjoitettuja ohjelmia. JUnit onkin eräs ilmentymä xUnit-arkkitehtuurista. JUnitin alkuperäiset tekijät olivat Erich Gamma ja Kent Beck, joista edellinen on tunnettu suunnittelumallien kehittäjä ja jälkimmäinen tunnetaan parhaiten testauslähtöisen kehitysmallin kehittäjänä. Jälkeenpäin moni muukin on osallistunut JUnitin kehittämiseen. [Jun05]

Automatisoinnin hallintaan päätettiin valita Maven. Maven on kehitetty tarpeesta ymmärtää ohjelmistoprojektien tilanne kulloinkin ja hahmottaa ohjelmistoprojekti kokonaisuutena. Maven helpottaa nimenomaan kehittäjien elämää, vaikka se voi olla avuksi päällikkö- ja johtajatason henkilöillekin. Työkalu pusertaa jo olemassa olevasta projektista erilaisia raportteja, mitattavia suureita, rakenteita, kaavioita ja dokumentteja julkaistaviksi. Toisaalta Mavenia voidaan käyttää projektien alussa luomaan pohjaa, kuten vaikkapa alustava hakemistorakenne, kehittäjien työlle. Maven on Antin tapaan Java-pohjainen ja sellaisena käyttöjärjestelmästä riippumaton. [Mav05]

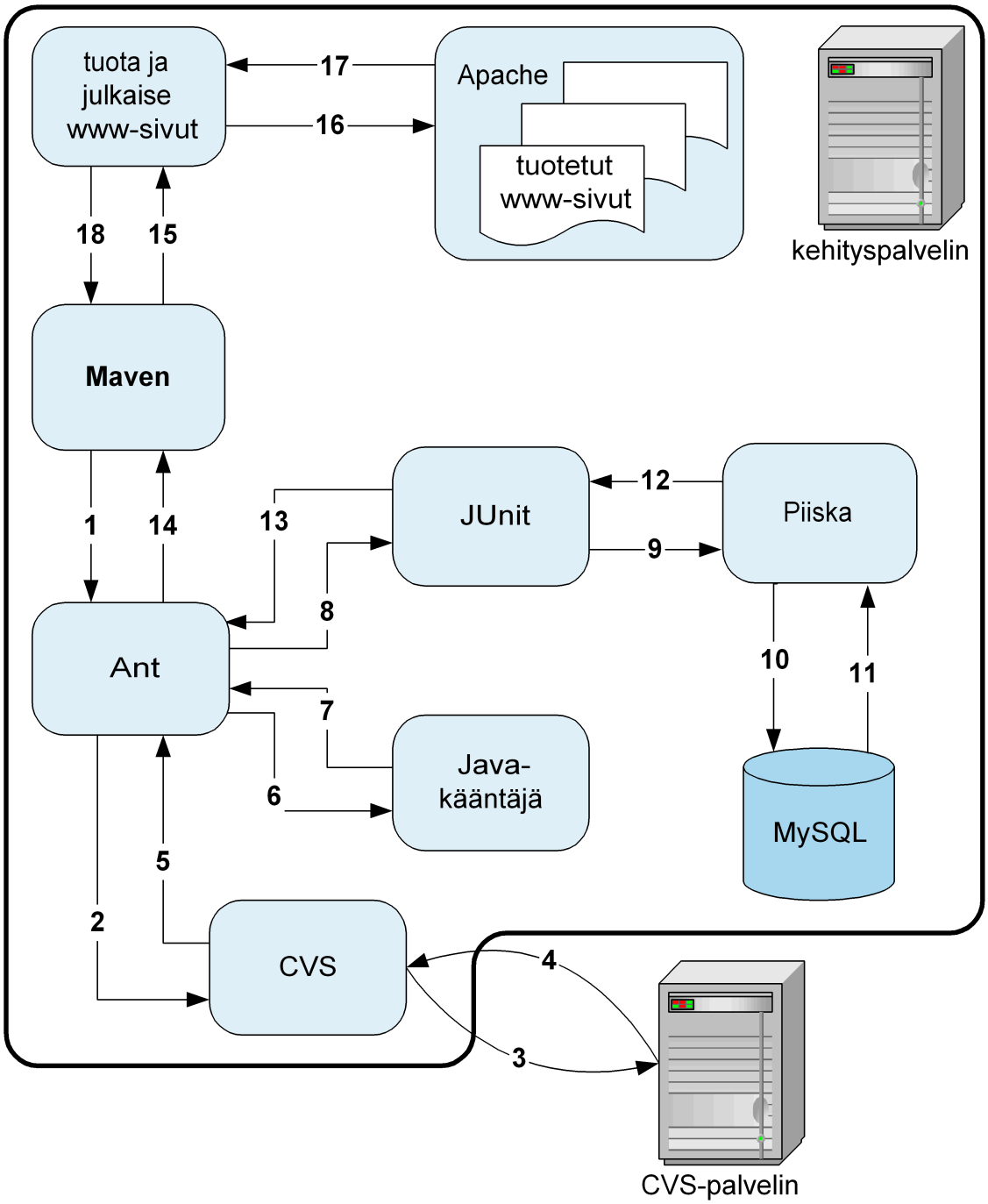
Testien tulokset julkaistaan Mavenin avulla www-sivustoksi yrityksen sisäiseen verkkoon kehittäjien saataville. Www-sivuja varten tarvittavaksi HTTP-palvelimeksi yrityksen tekninen ylläpito halusi käyttöön Apachen [Apa04]. Apache, Ant ja Maven ovat kaikki Apache Software Foundation -säätiön kehittämiä työkaluja. Apache-säätiön ohjelmiin luotetaan laajalti, ja näitä ohjelmia on käytetty useammassa alan kirjassa esimerkkeinä.

Testipalvelimen käyttöjärjestelmä on FreeBSD, eräs versio Unixista [Fre05]. Tämäkin oli teknisen ylläpidon tekemä valinta, vaikkakaan kenelläkään ei ollut muita ehdotuksia tai perusteita valintaan. Muut työkalut ovat joka tapauksessa joko käyttöjärjestelmästä riippumattomia tai saatavilla usealle käyttö-järjestelmälle, joten käyttöjärjestelmää voitaisiin haluttaessa vaihtaa. Unixin cron-ohjelmaa käytetään automaattisen prosessin ajastettuun käynnistämiseen.

Cron on kätevä valinta, sillä se tulee FreeBSD:n mukana ja on pitkään käytössä ollut luotettava ohjelma.

3.3 Automatisoitu testaus

Koostamisen, testaamisen ja raportoinnin automatisoitu prosessi voi alkaa täysin automaattisesti ajastettuna tai vaihtoehtoisesti prosessointikäsky voi tulla kehittäjältä. Kummassakin tapauksessa prosessi käynnistyy Mavenista. Kuvassa 5 havainnollistetaan järjestelmää.



1) Kun Maven ajetaan kääntämään ja testaamaan Piiska, se herättää Antin.
2-5) Ant hakee ensin Piiskan CVS-moduulin keskustietovarastopalvelimelta.
6-7) Seuraavaksi Ant ajaa lähdekoodin Java-kääntäjän läpi.
8-9) Antin JUnit-käsky ajaa Piiskan yksikkötestit.
10-11) Piiska käyttää tietokantaa.
12-14) JUnit-raportointityökalu antaa eteenpäin Piiskalla ajettujen testien tulokset.
15) Maven käyttää useita ohjelmalisäkkeitä www-sivujen tuottamiseen.
16) Tuotettu sivusto siirretään haluttuun paikkaan, jonka jälkeen sivusto on katseltavissa Apache-palvelimen kautta.
17-18) Maven saa tiedon prosessin onnistumisesta, ilmoittaa siitä käyttäjälle ja sulkeutuu.

Kuva 5. Automatisoitu testaus

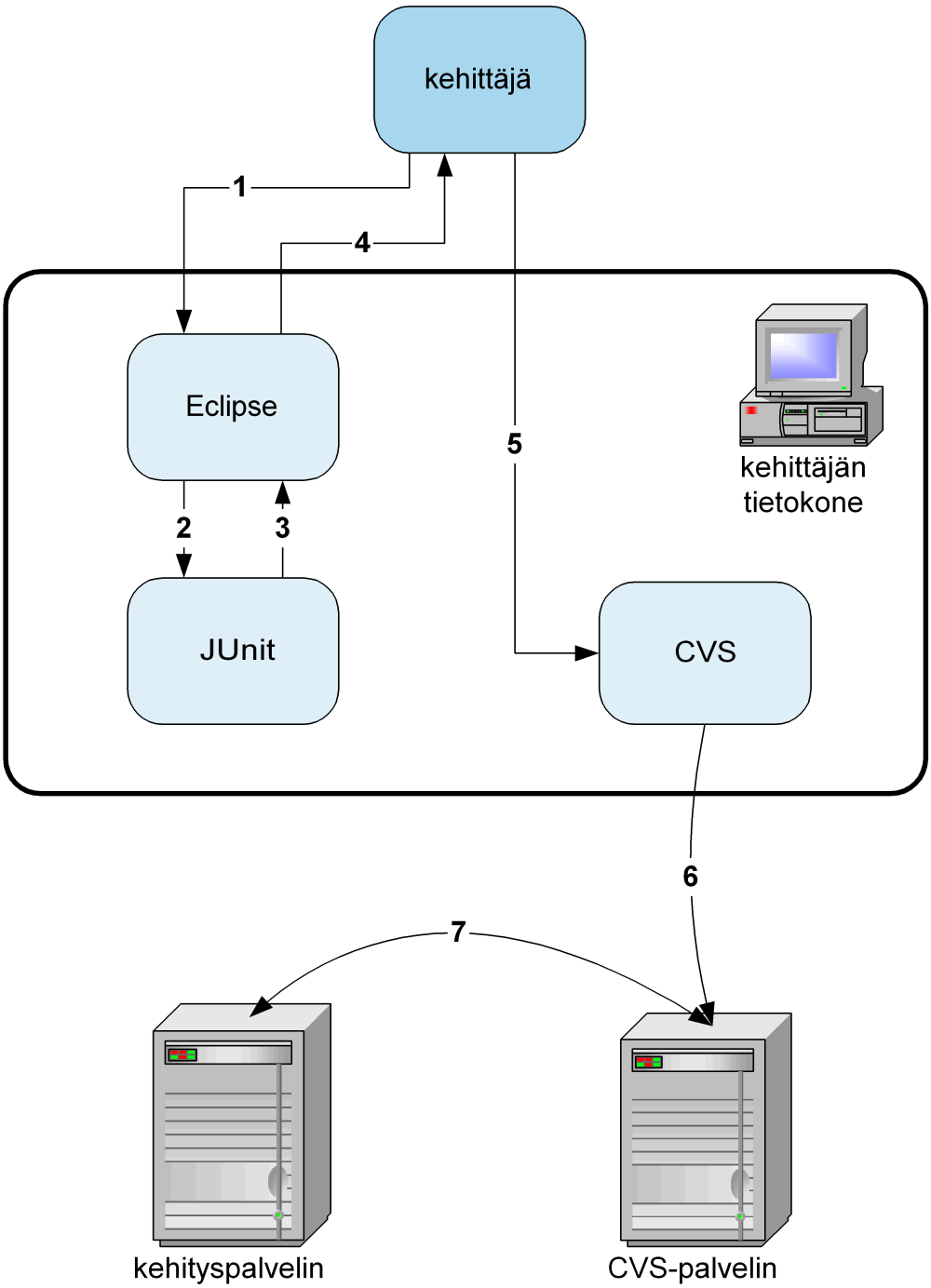
Mavenin arkkitehtuurissa eri toiminnot ovat ytimen päälle tehtyjä ohjelmalisäkkeitä (*plug-in*). Lisäkkeistä eniten käytetyt ja vähän virallisemmat tulevat ohjelman mukana. Piiska-projektia varten päätettiin Maveniin tehdä oma ohjelmalisäke. Tämä yksinkertaistaa asioita kehittäjän näkökulmasta, sillä kun toiminnot on pakattu yksiin kuoriin, kaikkien ei tarvitse tietää, mitä lisäominaisuuksia tarvitaan milloinkin ja missä järjestyksessä. Yksinkertaisimmillaan projektikohtainen ohjelmalisäke voi olla sellainen, että se piilottaa projektin mukaan nimetyn käskyn alle järkevään järjestykseen käskyt ajaa valmiita lisäkkeitä. Suurin osa valmiista ohjelmalisäkkeistä käyttää kuitenkin toinen toisiaan, mikä saattaa aiheuttaa jo parinkin käskyn ketjussa useita samojen vaiheiden suorituksia. Monimutkaisemmin toteutettu lisäke hyödyntää tilaisuutta poistaa ylimääräiset ja tarpeettomasti toistuvat vaiheet. Testaajan itse toteuttamaa ohjelmalisäkettä voidaan helposti muunnella tilanteen mukaan vastaamaan projektin kulloistakin vaihetta. Lisäksi omaan palikkaan voi olla hyvä alustaa joitakin järjestelmäkohtaisia harvoin muuttuvia arvoja. Lisäkkeiden tekemiseen käytetään Mavenissa XML-pohjaista skriptikieltä Jellyä. Mavenissa Jelly on läpinäkyvä kerros niin, että halutessaan Mavenia voi skriptata suoraan Antin käyttämällä skriptikielellä. [Mav05; Hig04]

Mavenin avulla ajetaan Ant koostamaan (*build*) ohjelma halutulla tavalla. Antin käsketään ensin hakea projektin moduuli versionhallinnan keskustietovarastosta, joka sijaitsee omalla palvelimellaan. Sitten Ant käyttää olemassa olevaa Java-kääntäjää kääntämään luokat, sekä varsinaiset ohjelmaluokat että myös testiluokat. Yksikkötestit ajetaan Antin kautta JUnitin tarjoamissa kehyksissä. [Ant05; Jun05; Hig04] Tieto testien sujumisesta otetaan talteen ja Mavenin erilaisia mukana tulevia ohjelmalisäkkeitä käytetään raporttien luomiseen ja prosessoimiseen ihmisille mukavaksi luettavaksi HTML-muotoisiksi www-sivuiksi. Kun kaikki halutut dokumentit on tehty, Maven siirtää ne palvelimella sellaiseen paikkaan, että ne ovat luettavissa Apachen

HTTP-palvelimen välityksellä kehittäjän selaimesta käsin. Liitteessä 1 on osa Mavenin tuottamasta Piiska-projektin www-sivustosta.

3.4 Kehittäjän työympäristö

Ohjelmoijat ja testaaja työskentelevät keskenään samankaltaisissa ympäristöissä. Näin viestintä projektiryhmän sisällä helpottuu. Kaavio työympäristöstä on kuvassa 6. Sovelluskehitysympäristönä käytetään Eclipseä [Ver05]. Eclipse on avoimen lähdekoodin ohjelma, jolle on saatavissa paljon ohjelmalisäkkeitä [Ecl05]. Ohjelmassa on sisäänrakennettuna CVS-version-hallinnan tuki ja graafinen JUnit-työkalu. JUnit on lähinnä testaajan käyttämä työkalu, mutta ohjelmoija saattaa tarvita sitä joissakin tilanteissa. Kehittäjä kirjoittaa ohjelmaa Eclipsellä, saattaa haluta ajaa joitakin testejä paikallisesti omalla koneellaan, ja kun ohjelma on kuosissaan, se tallennetaan keskustietovarastoon erilliselle palvelimelle. Tietovarastopalvelimelta muutokset lähdekoodiin päätyvät viimeistään automaattisen testausprosessin kautta kehityspalvelimelle. Muut kehittäjät saavat muutokset päivittäessään omia tiedostojaan.



1) Kehittäjä käyttää Eclipseä sovelluskehitysympäristönään.
2-4) Yksikkötestit voidaan ajaa suoraan Eclipsestä ja tulokset näkyvät heti Eclipsen graafisen käyttöliittymän kautta ruudulla.
5-6) CVS-versionhallintaohjelmaa käytetään - tarvittaessa sovelluskehitysympäristön ulkopuolelta - tallentamaan lähdekoodi keskustietovarastopalvelimelle.
7) Automaattinen koostusjärjestelmä kehityspalvelimella hakee muutokset haluttaessa itsenäisesti CVS-palvelimelta.

Kuva 6. Kehittäjän työympäristö

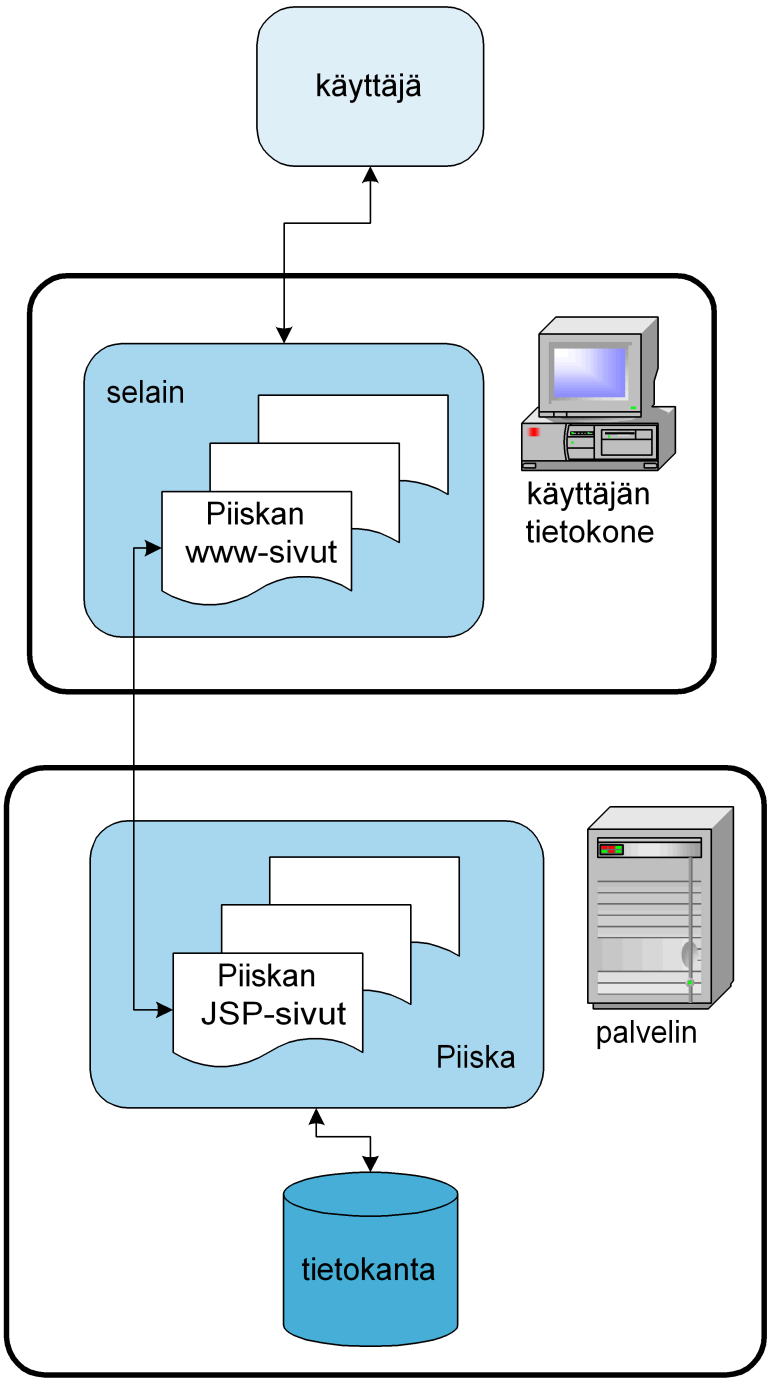
4 Piiska-sovelluksen testaus

4.1 Piiska-projekti

Työlainsäädännön mukaan yrityksillä täytyy olla virallinen ja kunnollinen työtuntiseuranta. Kun henkilöstön määrä Fathammerilla kasvoi ja työajan seuranta kävi hankalaksi muilla keinoin, päätettiin tehdä projektihallinta- ja työtuntikirjanpito-ohjelmisto Piiska. Piiskan tarvetta lisää yrityksen toiminnan painopisteen siirtyminen pitkäjänteisestä väliohjelmistotuotannosta pelitalon useiden yhtäaikaisten lyhyiden projektien hallinnoimiseen. Johdon ja esimiesten pitää kyetä perustamaan ajankulun arviot tosiasioihin, jotta budjetointi onnistuisi ja laskutus olisi oikeudenmukaista. Yrityksen toiminnan kanssa kasvoi samalla tarve tietää tarkemmin kuin ”musta tuntuu” -pohjalta, paljonko aikaa todellisuudessa menee eri tehtäviin ja projekteihin.

Piiska-ohjelmisto oli jo suurimmaksi osaksi toteutettu, kun tätä insinöörityötä aloitettiin. Insinöörityöhön kuului Piiskan testaaminen ja testisuunnitelman kirjoittaminen. Piiskan testaustyössä pyrittiin soveltamaan alustavasti kehiteltyä testauksen prosessimallia mahdollisuuksien mukaan ja näin koettelemaan mallin sopivuutta yrityksen projekteihin.

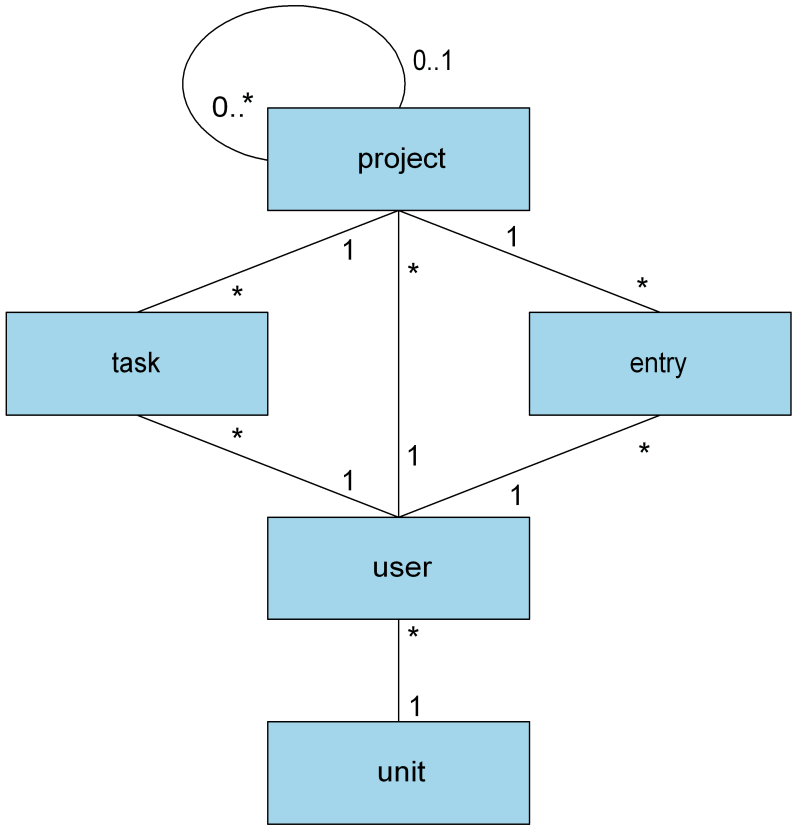
Piiska on projektihallinta- ja työtuntikirjanpito-ohjelma. Yrityksen projektipäälliköt käyttävät Piiskaa projektien seurannan, hallinnoimisen ja budjettisuunnittelun työkaluna. Projektipäälliköt merkitsevät Piiskaan alkavan projektin ja osoittavat projektiryhmän jäsenille projektiin kuuluvat tehtävät. Projektiryhmien jäsenet kirjaavat Piiskaan päivittäin kuhunkin tehtävään käyttämänsä työtunnit. Kaaviokuva järjestelmästä loppukäyttäjän kannalta on kuvassa 7. Piiska on palvelin-asiakasohjelmisto. Työntekijät käyttävät Piiskaa yrityksen sisäisessä verkossa www-pohjaisen käyttöliittymän lävitse, jota kehitetään käyttäjien toiveiden mukaisesti myöhemmässä vaiheessa. Kehittäjiä ja ylläpitäjiä varten Piiskassa on myös tekstipohjainen käyttöliittymä.



Kuva 7. Piiskan arkkitehtuurin yleiskuva

Kuvassa 8 on esitetty Piiskan ER-kaavio, joka havainnollistaa sitä, millaisia tietoja järjestelmään voi syöttää ja niiden suhteita toisiinsa. Projektiin (*project*) voi kuulua useita aliprojekteja. Yhteen projektiin voi kuulua useita tehtäviä (*task*) sekä ajankäyttömerkintää (*entry*). Yhdellä ohjelmiston käyttäjistä (*user*)

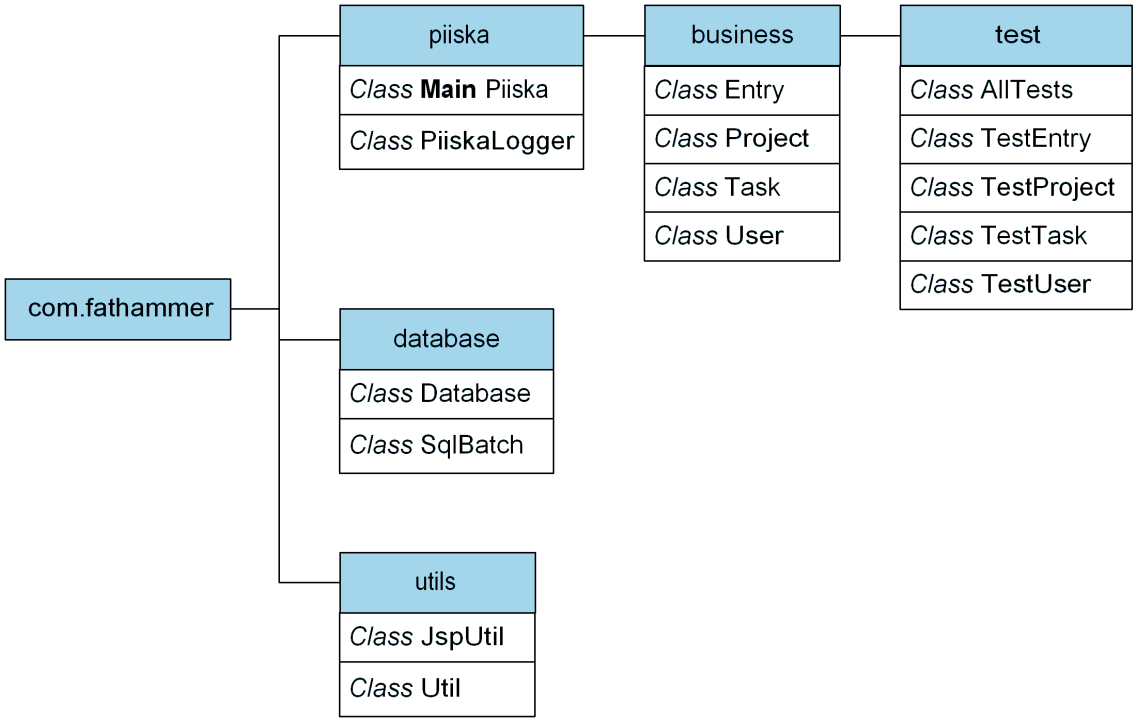
voi olla yhtäaikaisesti useita tehtäviä, ja hän voi tehdä useita ajankäyttömerkintöjä. Yksi henkilö voi kuulua useaan projektiin, mutta vain yhteen yrityksen osastoon (*unit*). Järjestelmän tiedot tallentuvat tietokantaan. Kantakyselyissä on käytetty standardia ANSI-SQL:ää, joten ohjelmisto on riippumaton tietokannan valmistajasta. Tietokantaohjelmaksi oli valittu MySQL.



Kuva 8. Piiskan ER-kaavio

Palvelinohjelmisto on toteutettu Javalla ja JSP-sivuilla (*Java Server Pages*). JSP:llä tehdään käyttöliittymäkerros loppukäyttäjille. Varsinainen ohjelma liiketoimintalogiikkoineen ja kantayhteyksineen on perus-Javaa, mutta JSP-sivutekniikka kuuluu Javan J2EE-versioon (*Enterprise Edition*). Piiska itsessään on itsenäisesti ajettava (*stand-alone*) ohjelma, mutta JSP-sivut tarvitsevat toimiakseen sovelluspalvelimen (*application server*), kuten esimerkiksi Apache Jakarta Tomcatin. Piiskan luokkakaavio on esitetty kuvassa 9.

Piiskassa on käyttäjätunnus- ja salasanavarmennus. Järjestelmän ylläpitäjä luo käyttäjätunnuksen ja syöttää käyttäjän haluaman salasanan kerta-luonteisesti. Tietoturva jätetään merkittäviltä osin yrityksen verkkoylläpitäjien varaan.



Kuva 9. Luokkakaavio Piiskasta

4.2 Yksikkötestaus

Kehiteltyä prosessimallia soveltaen Piiskalle tehtiin perusteelliset yksikkötestit metoditasolla. Testien läpimenoasteeksi vaadittiin täydet 100 %. Piiskasta testattiin vain ydinliiketoimintalogiikka, joka on riippumaton käyttöliittymän toteutuksesta [NLL05]. Liiketoimintalogiikan testaaminen testaa samalla näiden luokkien käyttämät, erilaisia palveluja, kuten kantayhteyksiä, tarjoavien luokkien metodit. Testattaessa ei käytetty harjoituskohteita tietokanta-operaatioiden testaamiseen, vaan ohjelma testattiin tuotanto-ympäristöä vastaavassa testiympäristössä [Ver05]. Tätä varten Piiskalle luotiin yksi tietokanta vain testausta varten. Piiskasta testatut luokat ja testiluokkien sijainti näkyvät kuvassa 9. Testiluokat ovat testattavan paketin test-nimisessä

alipaketissa, ja ne on nimetty etuliitteellä Test testaamaansa luokkaa vastaavasti [Ast03]. Testipaketissa on myös testisarjaluokka AllTests.

Kuvassa 10 on esitetty esimerkkeinä luokkien sisällöstä Entry-luokan tärkeimmät metodit ja luokalle tehdyn testiluokan TestEntryn vastaavat testimetodit eli testitapaukset (*test case*). Kaaviossa ei näy Entryn yksinkertaisia käpeltimiä (*accessor*), joita ei ole tarpeellista testata. TestEntry-luokan metodikaavioon on merkitty testimetodien lisäksi JUnit-kehyksen rajapinnan vaatimukset [Jun05]. Kuvassa on myös kaikki paketin testit sarjaksi kokoava luokka AllTests. Esimerkkinä yksikkötestien ohjelmakoodista on liitteessä 2 listaukset luokista TestEntry ja AllTests. TestEntrystä on listattu vain otteita, mutta AllTests on esitetty kokonaisuudessaan.

Class Entry
public Entry()
public Entry(long)
public synchronized void delete()
public synchronized void insert()
public void update()
public Vector getAllEntries()
public Vector getAllUsersEntries(**long** userId)
public Vector getAllUsersEntries(**long** userId, **long** rootProjectId)
public Vector getProjectsEntries(**long** projectId)
public Vector getUsersEntries(**long** userId, **long** projectId)

Class TestEntry *extends* TestCase
public TestEntry(**String** nameOfTestMethod)
protected void setUp() *throws* Exception
protected void tearDown() *throws* Exception
public void testEntry()
public void testDeleteEntry()
public void testInsertEntry()
public void testUpdateEntry()
public void testGetAllEntries()
public void testGetAllUsersEntriesLong()
public void testGetAllUsersEntriesLongLong()
public void testGetProjectsEntries()
public void testGetUsersEntries()

Class AllTests
public static Test suite()

Kuva 10. Luokkien määrittelykaavio

JUnit-kehys on julkaistu omassa paketissaan nimeltä junit.framework. Kehystä käytettäessä testiluokka perii luokan TestCase edellä mainitusta paketista.

Testiluokan konstruktori, joka ottaa argumentikseen String-tyyppisen muuttujan, vaadittiin JUnitin 3.7-versioon saakka. Se muuttui vapaaehtoiseksi – muttei vanhentuneeksi (*deprecated*) – versiossa 3.8, jota käytettiin Piiskan testauksessa. Vanhantyyppinen konstruktori sisällytettiin Piiskan testiluokkiin yhteensopivuuden takaamiseksi vanhempien versioiden kanssa. TestCasen perivän testiluokan täytyy sisältää ainakin yksi etuliitteellä test nimetty testimetodi, jotta testisarjaa rakentava suunnittelumalli kykenisi toimimaan. [Jun05]

Samassa testiluokassa on useita testimetodeja. Testimetodit tarvitsevat monesti keskenään samankaltaisen joukon muuttujia ja olioita, ennen kuin testit voidaan suorittaa. Tällaista alkuasetelmaa kutsutaan testipuitteiksi (*fixture*). JUnit-kehyksissä testipuitteet rakennetaan metodilla setUp. Testiluokissa syrjäytetään (*override*) perityn TestCase-luokan metodit setUp ja tearDown. Nämä metodit voi joko jättää tyhjiksi viitaten perityn luokan toteutuksiin tai sijoittaa puitteiden alustus setUpiin ja resurssien vapautus tearDowniin. TestEntry-luokassa on toteutettuna sekä setUp- että tearDown-metodit, sillä Entry-luokan metodit tarvitsevat ainakin yhden varmasti olemassa olevan Project-luokan olion. Tärkeintä on, että metodin setUp toteutuksessa alustetaan ja luodaan ainakin oliot, jotka tehdään ohjelman omista luokista. Tämä siksi, että mikäli testin suorittaminen kaatuu johonkin muuhun ohjelman osaan kuin itse testattavaan metodiin, halutaan virheilmoitus epäonnistumisesta muualla kuin testimetodissa ja ohjelman omien osien toiminta on epävarmempaa kuin Javan perusluokkakirjastojen. Metodissa tearDown tuhotaan väliaikaiset testiä varten luodut, mahdollisesti jäljelle jäävät oliot, jotka setUpissa luotiin, ja vapautetaan siinä varatut resurssit. [BuC03; Bec02]

JUnit-kehyksessä jokaiselle testimetodille kutsutaan erikseen setUp ennen testimetodin ajamista ja tearDown testin ajon jälkeen – riippumatta testin onnistumisesta. Tämä hidastaa testien suorittamista, mikäli setUpissa joudutaan luomaan kovin raskaat puitteet. Puitteet voidaan luoda myös vain kerran useammalle testimetodille, jolla kyetään tarvittaessa optimoimaan

testien ajoaikaa. Piiskan tapauksessa tällaiselle optimoinnille ei ollut tarvetta. Koko projektin kaikkien testien ajo vei vain noin 15 sekuntia, vaikka suurin osa testeistä loi tietokantayhteyden yhä uudestaan. [Bec02; BuC03]

Testimetodit ovat yksittäisiä testitapauksia, joissa testattavan asian tulee olla binäärinen eli testin lopputulos on johdettavissa totuusarvoksi. Metodit ovat julkisia ja alkavat etuliitteellä test. Puitteita vasten metodissa suoritettavan ohjelmakoodin lopuksi on yksi tai useampi tarkistus (*assert*), joissa todennetaan kahden muuttujan sisällön suhde, kahden olion tila tai viittaukset olioihin. Mikäli tarkistuksessa tilanne on sellainen kuin on haluttu, testi onnistuu ja menee läpi. Jos taas tilanne tarkistettaessa on jotenkin toivotusta poikkeava, JUnit-kehys hylkää (*fail*) testin. Epäonnistumisen (*failure*) varalta tarkistukset tehdään, joten niitä on odotettavissa. Testin ajaminen lopetetaan saman tien ensimmäiseen tarkistuksen epäonnistumiseen. Jos testissä on useampi tarkistus, epäonnistuneen tarkistuksen jälkeiset tarkistukset jäävät siis tekemättä. Esimerkiksi liitteessä 2 TestEntry-luokan testimetodissa testUpdateEntry on kaksi tarkistusta, assertEquals(), joista alempaa ei tehdä ylemmän epäonnistuessa. Testien suorittaminen saattaa keskeytyä myös odottamattomasti. Tällöin JUnit-kehys palauttaa testistä virheen (*error*) ja testin ajo loppuu. [BuC03]

Luokka AllTests kokoaa Piiskan liiketoimintalogiikan testipaketin kaikki testit yhdeksi testien sarjaksi. AllTests sisältää vain yhden metodin, jossa Piiskan tapauksessa annetaan JUnit-kehyksen erottaa annetuista testiluokista automaattisesti kaikki julkiset test-alkuiset metodit. JUnit käyttää tähän heijastus-suunnittelumallia (*reflection design pattern*) [BuC03; OCW01]. Toinen vaihtoehto on määritellä yksitellen kaikki sarjaan sisältyvät testimetodit, mikä käy vaivalloiseksi useiden testiluokkien ja lukuisten testimetodien tapauksessa. Sarjan rakentavan metodin nimen täytyy olla suite. JUnit-kehyksen testiajaja (*test runner*) etsii nimenomaan sennimistä metodia [BuC03]. Myös Ant etsii metodia suite() ajaessaan testejä. Metodin tulee

palauttaa Test-rajapinnan toteuttava olio [Jun05]. Piiskan tapauksessa käytettiin JUnit-kehyksen tarjoamaa rajapinnan toteuttavaa TestSuite-luokkaa.

Yksikkötestejä varten rakennettiin luvussa 3 kuvattu automaattinen testausympäristö. Piiskan yksikkötestit ajettiin ajastetusti joka yö Mavenin avulla. Mavenin yöllisestä testiajosta tuottamalta sivustolta näki helposti rikkinäisten testien määrän. Testaaminen lopetettiin, kun kaikilla testattavaksi halutuilla luokilla metoditasoisten testien kattavuus oli riittävä ja testien läpimenoaste täydet 100 %. Liitteessä 1 on Mavenin tuottamien sivujen näkymä sen jälkeen, kun testaus oli valmis.

4.3 Muu testaus ja testaussuunnitelma

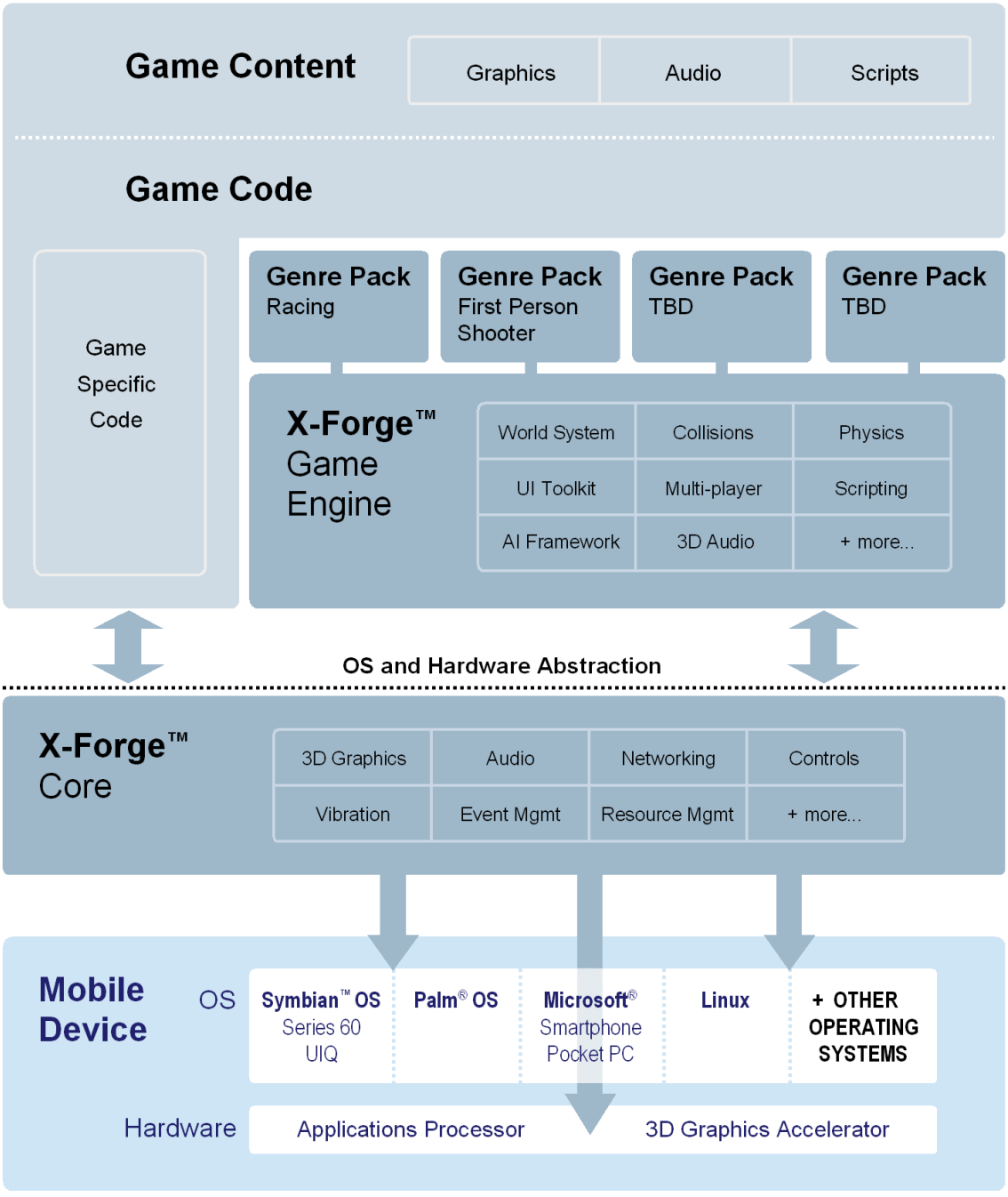
Piiska on tarkoitettu vain yrityksen sisäiseen käyttöön. Käyttäjämäärät ovat siksi tiedossa ja kohtuullisen vähäisiä, vaikka kaikki päättäisivät käyttää Piiskaa yhtäaikaa. Piiskan toimintakyky lyhyellä aikavälillä ei ole kriittinen yritykselle. Näistä syistä suorituskykytestaukset jätettiin pääosin tekemättä. Järjestelmää korjataan tarpeen mukaisesti, mikäli suorituskyky osoittautuu riittämättömäksi. Turvatestauksia ei tehdä – ellei ongelmia ilmene – sillä voidaan olettaa, että omat työntekijät ovat ystävällismielisiä Piiskan käyttäjiä. Tekninen ylläpito huolehtii Piiskan tietoturvasta verkon sisällä. Yrityksen sisään jäävästä, suppean käyttäjäkunnan ohjelmasta ei ole tarpeellista tehdä laajoja käytettävyystestejä. Lisäksi korjaustoiveet kulkevat yrityksen sisällä Request Tracker -tehtävänantojärjestelmän avulla nopeasti, jos käyttöliittymässä on pahoja vikoja ja ohjelmiston kehittäjät ovat lähellä korjaamassa sovellusta. Näin ollen käytettävyystestausta ei toteutettu lainkaan.

Piiskalle tehtiin IEEE:n standardia 892-1998 mukaillen testaussuunnitelma [IES98a]. Testaussuunnitelma on liitteessä 3. Osa standardin mukaan dokumenttiin kuuluvista kohdista on jätetty kirjoittamatta yritykselle tarpeettomina tai projektin luonteeseen sopimattomina. Näistä kohdista on dokumenttiin kuitenkin kirjattu otsikot, jotta niiden alle voidaan helposti tarvittaessa lisätä tekstiä. [Ver05]

5 X-Forge Online -järjestelmän testaussuunnitelma

5.1 X-Forge -teknologia

Fathammerin aiemmat pelit on tehty yrityksen kehittämälle väliohjelmistolle nimeltä X-Forge. X-Forge on rakennettu siten, että 3D-pelikoneen ydin tehdään jokaiselle tuetulle mobiililaitteelle erikseen laitteistokohtaisella teknologialla. Väliohjelmiston päälle voidaan tehdä itse peli, joka toimii periaatteessa kaikilla laitteilla, joita X-Forge tukee. Järjestelmän kaavio näkyy kuvassa 11. Väliohjelmiston ja sille tehtyjen pelien kehityskielenä on ollut C++.



Kuva 11. X-Forge-teknologia [Fat05]

Fathammerin strategiamuutoksen takia X-Forgea ei ole enää tarkoitus lisensoida ulospäin, mutta sitä tullaan käyttämään edelleen sisäisesti ja yhteistyöyritysten kanssa pelialustana. X-Forgea kehitetään jatkossa vain omiin tarpeisiin. Tähän mennessä X-Forge on tukenut vain lähiverkkoratkaisuja.

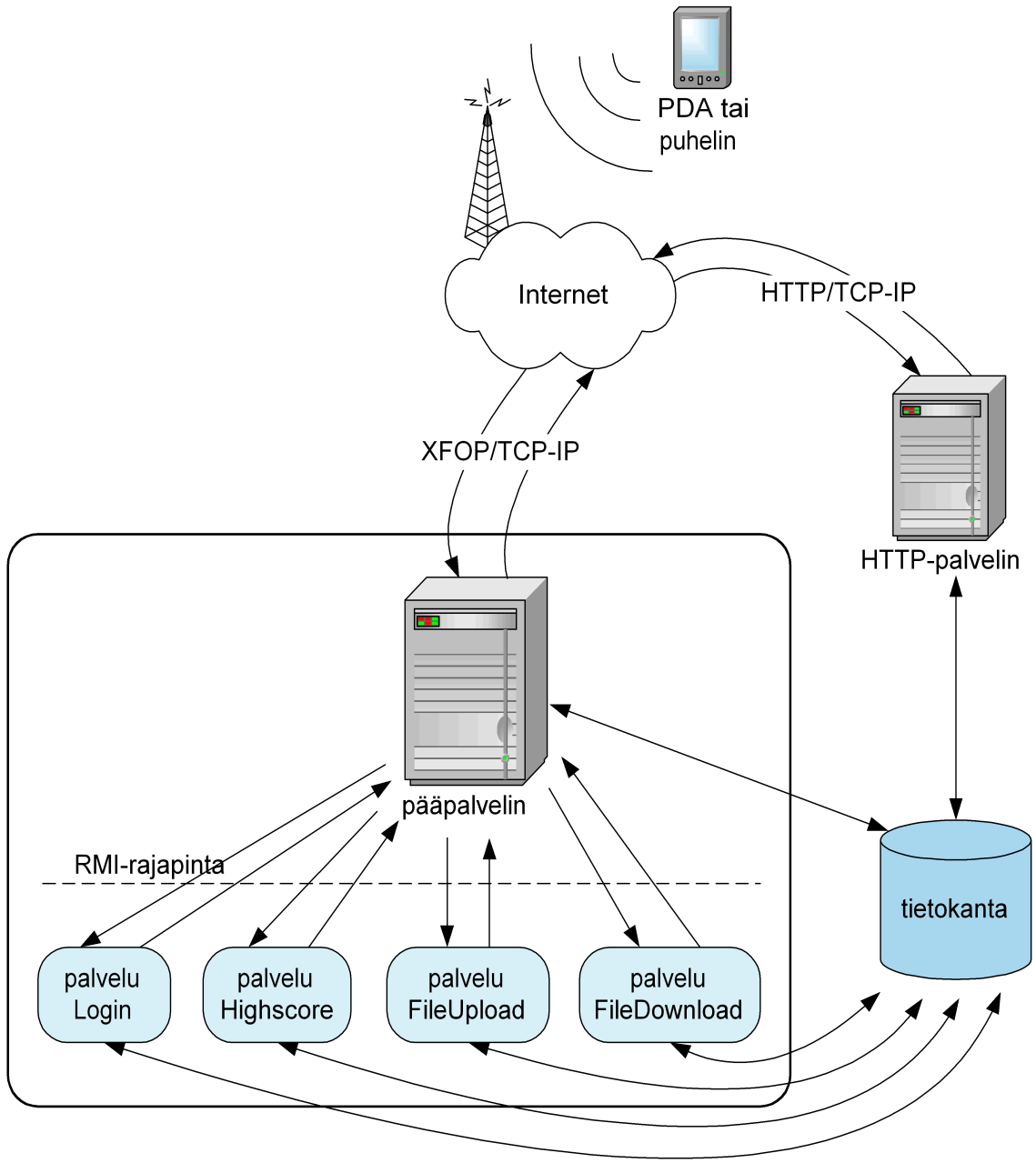
Toteutettuja verkkotyyppejä on kaksi, vertaisverkko (*peer-to-peer*) ja paikallispalvelin-asiakasverkko (*client-server*). Ensimmäistä käytetään kahden pelaajan välisessä verkkopelissä, ja jälkimmäisessä useamman pelaajan verkossa yksi laite ottaa itselleen palvelimen roolin. Näissä lähiverkoissa laitteiden etäisyys toisistaan on voinut olla korkeintaan muutamia kymmeniä metrejä. Uutena osana väliohjelmistoon rakennetaan X-Forge Online (*XFO*). Verkkoyhteyden mahdollisuutta päätelaitteiden välillä laajennetaan Internetin välityksellä laajaverkoksi X-Forge Onlinen pääpalvelin-asiakasarkkitehtuurin avulla. XFO:n myötä voidaan toteuttaa pelille vaikkapa maailmanlaajuinen parhaiden pelaajien lista ja antaa pelaajalle mahdollisuus pelata kaverin kanssa toiselta puolelta Suomea. Lisäksi pelaaja voi XFO:n välityksellä vaikkapa hakea uutta sisältöä peliinsä.

Insinöörityön yksi tarkoitus oli tehdä testaussuunnitelma X-Forge Onlinelle. X-Forge Onlinen tarve oli juuri tiedostettu insinöörityötä aloiteltaessa. Kun Piiskan testaustyö oli valmis, XFO:n pääarkkitehtuuri ja toteutustavat olivat selvenneet, mutta projekti eli koko ajan edelleen. X-Forge Online on hyvä esimerkki siitä, kuinka perinteiset ohjelmistokehityksen prosessimallit yksinään olisivat olleet yrityksen käyttöön aivan liian jäykkiä. Piiskasta saatujen kokemusten mukaan testauksen prosessimallia hiottiin XFO:n testaus-suunnitelmaan. Testauksen toteuttaminen ei kuulu insinöörityön alueeseen.

5.2 X-Forge Online -projekti

X-Forge Onlinesta tehdään pelien ja pelaajien vuorovaikutukseen tarkoitettu Internetin lävitse toimiva asiakas-palvelinohjelmisto. Kuva 12 havainnollistaa X-Forge Onlinen arkkitehtuuria. Asiakaspuolen ohjelma tulee toimimaan laitteistoltaan peleihin kykenevissä mobiileissa päätelaitteissa, kuten PDA:t ja tehokkaat matkapuhelimet. Asiakasohjelma toteutetaan C++-ohjelmana. Asiakas- ja palvelinohjelmisto keskustelevat keskenään, mikäli päätelaitteessa on toimiva Internet-yhteys. Viestintään käytetään X-Forge Onlinea varten suunniteltua pakettimuotoista protokollaa, XFOP, joka liikkuu Internetissä TCP-IP:n välityksellä. XFOP-protokollaa suunniteltaessa tärkeimmät asiat ovat

olleet keveys ja mukautuvuus. Mukautuvuutta tarvitaan pelien todella vaihtelevien tarpeiden vuoksi ja keveyttä mobiililaitteiden hyvin rajallisten muistiresurssien ja verkkokapasiteetin takia. Palvelinpuolella ohjelmisto toteutetaan Javalla. Tässä insinöörityössä rajoitutaan tarkastelemaan palvelinohjelmistoa.



Kuva 12. X-Forge Onlinen arkkitehtuuri

Palvelinohjelmisto on palvelujärjestelmä (*front end system*). Ydinarkkitehtuuri koostuu pääpalvelinohjelmasta ja erilaisista palveluista sen taustalla. Pääpalvelinohjelma on yksinkertainen. Se on tarkoitettu lähinnä oikeuksien tarkistamiseen ja liikenteen ohjaamiseen palveluille. Palvelut toimivat pääpalvelimen kanssa RMI-rajapinnan (*Remote Method Invocation*) läpi, ja niitä ajetaan eri virtuaalikoneessa kuin pääpalvelinohjelmaa. Näin palvelut voidaan tarvittaessa hajauttaa helposti usealle fyysiselle palvelinlaitteistolle. Kaikki palvelut, pääpalvelin ja tiedostojen siirtoon omistettu eriytetty HTTP-palvelin käyttävät tietokantaa. Tietokannassa ylläpidetään paitsi ohjelmien tarvitsemia staattisia tietoja, kuten käyttäjänimiä, niin myös dynaamisempaa tietoa, esimerkiksi aktiivisten istuntojen tilaa koskevia.

X-Forge Online tehdään yrityksen sisäiseksi työkaluksi ja lisäksi Fathammerin yhteistyökumppaneiden käyttöön. Loppukäyttäjiä ovat kuitenkin pelaajat. Ulkopuolisia muuttujia on paljon, joten käyttäjämääriä ei voi kunnolla ennustaa, mutta ne voivat kasvaa suuriksikin. On varauduttava siihen, että vaikka yhteistyökumppanit olisivat ystävällismielisiä, niin loppukäyttäjät eli pelaajat voivat olla ilkeämielisiä ohjelmiston käyttäjiä. XFO:n on tarkoitus tulla pysyvästi pystytetyksi avoimeen tietoverkkoon, jota kautta tulee aivan varmasti vihamielisiä hyökkäyksiä. Palvelinohjelmisto täytyy siis testata perusteellisesti tietoturvan ja vakauden kannalta. Testien pitää kattaa mahdollisimman laajalti myös erilaisten harvinaistenkin virhetilanteiden käsittely. [Ver05]

X-Forge Onlinen testaussuunnitelma tehtiin edellä selostetun yritykselle kootun testausprosessimallin mukaisesti IEEE:n dokumentointistandardi nro 892-1998 perustanaan [IES98a]. Testaussuunnitelma on kokonaisuudessaan liitteessä 4. Suunnitelman sisältö esitellään seuraavissa luvuissa. Suunnitelmaa tehtäessä käytettiin hyödyksi Piiska-sovelluksen testauksessa saatuja kokemuksia. Suurin osa kirjallisuudesta haetuista käytännöistä havaittiin hyvin käyttökelpoisiksi, joten Piiska-projektin käytäntöjä vain hiotaan ja laajennetaan.

5.3 Yksikkötestaus

Piiska-projektin yhteydessä kokeillut yksikkötestauksen käytännöt todettiin hyviksi, joten kehitellyn prosessimallin ja käytäntöjen soveltamista jatketaan XFO-projektissa. Yksikkötestauksen apuvälineenä tullaan käyttämään JUnitia. Testaaja kirjoittaa ja järjestää yksikkötestit testattavan paketin test-nimiseksi alipaketiksi, johon tulevat testiluokat. Testattavat alueet testataan metoditasolla. Ainakin kaikki julkiset ja jotkin suojellut metodit testataan. Konstruktoritkin testataan, mikäli niissä suoritetaan muuta kuin kenttien alustusta. Yksinkertaiset käpeltimet (*accessors*) jätetään testaamatta.

JUnit-testit tullaan erottelemaan omiksi luokikseen. Testiluokat nimetään samalla nimellä kuin testattava luokka, mutta etuliitteellä Test. Testiluokan sisällä on jokaiselle testattavan luokan testattavalle metodille testimetodi, joka nimetään vastaavasti samannimiseksi etuliitteenään test. Mikäli metodeja on ylikuormitettu, tehdään testimetodit kaikille metodeille erikseen ja testi-metodin perään kirjoitetaan parametrien tyypit järjestyksessä. Jokaisessa paketissa tulee olla AllTests-niminen luokka, joka käärii paketin testiluokat yhdeksi testien sarjaksi (*suite*).

Ohjelman sisältä, mutta testattavan luokan ulkopuolelta testien tarvisemien palvelujen kutsut tulee mahdollisuuksien mukaan sijoittaa testiluokan setUp-metodiin. Näin nähdään heti, jos testi on keskeytynyt johonkin muuhun kuin itse testattavan metodin pettämiseen. Haluttaessa raskaat resurssit ja muut järjestelyt voidaan luoda sarjalle testimetodeja vain kerran. Tätä käytetään vain tarkasti harkiten, sillä testien on tarkoitus olla mahdollisimman pitkälle eristettyjä. Kaikki rajoitetut resurssit, joita testimetodi on saattanut tarvita, vapautetaan tearDown-metodissa.

Testattaessa ei tulla käyttämään harjoituskohteita (*mock-objects*), vaan istutetaan ohjelmat alusta alkaen mahdollisimman tuotantoympäristön kaltaiseen testiympäristöön. Ohjelmistot täytyy joka tapauksessa testata aitojen riippuvuussuhteiden kanssa, joten ne voidaan luoda heti alussa. Muuten aikaa

hukkaantuu turhaan sellaisten testien kirjoittamiseen, jotka eivät heijasta todellisuutta. Tällä tavoin voidaan vältellä tuotantoympäristöön siirrossa vaanivia katastrofeja. [Ver05; NLL05]

Ohjelmistojen ulkopuolista dataa käsitteleville metodeille tullaan tekemään tavanomaisten yksikkötestien lisäksi tuhmat testit (*naughty tests*), joissa metodeille syötetään epämuodostunutta tai ilkeämielistä dataa. Mahdolliset kantayhteydet täytyy myös testata tällä tavoin tuhmasti. Yksikkötestit rakennetaan niin, että samalla testataan virhetilanteiden käsittelyä. Käytännössä osa testeistä antaa poikkeuksia tahallaan. Tällaiset kaatuma-harjoitukset (*crash test dummy pattern*) tehdään sisäisten luokkien avulla, jotka simuloivat virhetilanteen aiheuttavaa tilannetta. Yleensä kaatu-maharjoituksessa ei tarvitse luoda kokonaista harjoituskohdetta, vaan riittää, että siinä käytetty sisäinen luokka heittää testattavaksi halutun poikkeuksen. XFO:n kaltaisissa laajoissa ohjelmistoissa, joilla on suuri ja mahdollisesti vihamielinen käyttäjäkunta, on tärkeää testata tällä tavoin harvinaiset virhetilanteet. Testaajan on hyvä olla luovasti tuhoava, sillä vain testaajan kierous saattaa olla ainoa tapa löytää erikoiset ja epätavalliset virhetilanteet. [Bec02]

5.4 Integraatiotestaus

Integraatiotestausta tullaan tekemään jatkuvasti sitä mukaa, kuin ohjelman osia valmistuu. Ennen kuin luokka etenee integraatiotestaukseen, luokan kaikkien yksikkötestien tulee mennä läpi. XFO:n arkkitehtuurin vuoksi valmiit luokat integroidaan ensin siihen ohjelman osaan, johon ne kuuluvat, ja vasta sen jälkeen integroidaan yhteen eri ohjelman osat [Jef94]. Palveluun kuuluvat luokat integroidaan palveluun ja pääpalvelinohjelman luokat pääpalve-linohjelmaan. Sitten integraatio etenee palveluiden ja pääpalvelinohjelman välille.

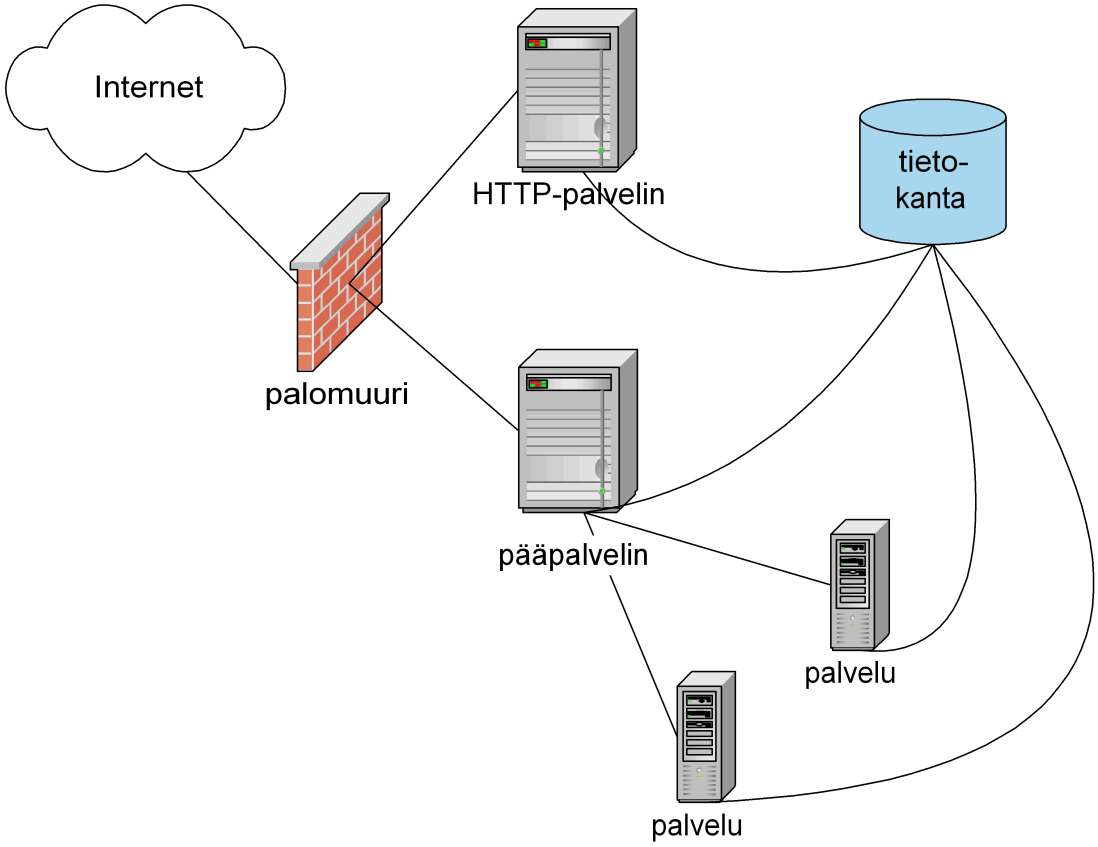
5.5 Kuormitus- ja suorituskykytestaus

Ohjelmiston suorituskyky täytyy testata, jotta saadaan selville, missä menee yhtäaikaisten käyttäjien määrän raja ja kuinka paljon järjestelmä kuormituksessa hidastuu. Pääpalvelimelle tehdään useilta koneilta lukuisia palvelupyyntöjä samaan aikaan. Tätä tullaan testaamaan yrityksen sisäisessä verkossa, jota todennäköisesti joudutaan päivittämään nopeampaan kapasiteetiltaan 1 Gb:n lähiverkkoon.

Integraatio- ja kuormitustestauksen apuvälineenä tullaan käyttämään JProbe-nimistä ohjelmistoa. JProbe on Quest Softwaren tekemä laaja analyysiohjelmisto testauksen eri osa-alueisiin. JProbe analysoi ohjelmia Java-virtuaalikoneesta saatavan tiedon avulla ja näyttää kerätyn tiedon käyttäjäystävällisessä muodossa, esimerkiksi graafeina. JProben analysoiva osa ohjelmaa kiinnittyy testattavaa ohjelmaa ajavaan virtuaalikoneeseen JVMPI:n (*Java Virtual Machine Profiling Interface*) kautta. [JPr04, s. 30—31]

5.6 Turvatestaus

Loppuvaiheen testauksissa kehityspalvelin järjestetään yrityksen sisäisessä verkossa tuotantovaiheen kaltaiseen verkkoympäristöön. Ohjelmisto ei ota kantaa verkon rakenteeseen tai palomuurin käyttöön. Ohjelmistot suunnitellaan ja rakennetaan niin, että palomuurin pystyttäminen on mahdollista. X-Forge Onlinen arkkitehtuuri pohjaa oletukseen kuvan 13 mukaisesta verkkotopologiasta. Palvelimet on tarkoitettu suojattavaksi palomuurilla. Verkossa pääpalvelin on vierekkäin palveluita tarjoavien palvelimien kanssa. Palveluita voi olla käynnissä useita samassa fyysisessä palvelinkoneessa, joka voi olla myös itse pääpalvelinkone. XFO:ssa on ulospäin näkyviä portteja vain yksi. Tämä helpottaa palomuurin pystyttämistä ja verkon ylläpitoa. Ohjelmiston palvelujen välinen liikenne hoituu palveluille läpinäkyvästi RMI-arkkitehtuurin lävitse. Tietokantaliikenne kulkee JDBC-ajureiden välityksellä. Haluttaessa myös tietokanta voidaan suojata palomuurilla. Tieto liikkuu verkon rakenteesta riippumatta, kunhan tietokannan porttiin sallitaan liikenne.



Kuva 13. X-Forge Onlinen suunniteltu verkkotopologia

X-Forge Onlinen ensimmäisessä toteutuksessa ohjelmistoa ei tehdä erityisesti kestämään DDoS-hyökkäyksiä (*Distributed Denial of Service*). Suojan ohjelmoiminen kasvattaa reippaasti ohjelman monimutkaisuutta, ja hyökkäyksiltä on silti vaikea suojautua. Riittävät testit DDoS:n varalta tulevat tehdyiksi kuormitustestien yhteydessä. Niissä tullaan saamaan tieto järjestelmän kyvystä selvitä kuormittavista tilanteista ja voidaan arvioida riskiraja yhtäaikaisille palvelupyynnöille. [NLL05]

6 Yhteenveto

Insinöörityössä saatiin kehitettyä Fathammer Oy:lle räätälöity testausprosessimalli Java-kehitykseen. Prosessimallin suunnittelu antoi yritykselle kaivattua syvempää tietoa Extreme Programming -metodologiasta sekä perusteita yrityskulttuuriin kasvaneelle vapaamuotoiselle kehitystyylille. Toisaalta yrityksen testausprosessimallia hieman formalisoitiin perinteisen ohjelmistokehitysmallista otetun dokumentointimallin avulla. Prosessimallista tuli hyvin yrityksen tarpeita vastaava. Suunniteltu testausprosessimalli myös mukautuu helposti muuttuvien projektien ja epävakaan liiketoimintailmaston mukaan. Tietoa erilaisista prosessimalleista ei ollut vaikea löytää, joten testausprosessimallin kokoaminen ei ollut ongelmallista.

Kehitetty testausprosessimalli perustuu pitkälti käytännön testausympäristön automatisointiin. Ilman mahdollisimman pitkälle vietyä automatisointia testauksen taakka kasvaa liian suureksi, jotta testausprosessimallia kyettäisiin seuraamaan. Suurimmat vaikeudet insinöörityössä liittyivät yksikkötestausympäristön automatisointiin. Suurin osa raskaan sarjan kehitysympäristöistä on yleensä Unix-pohjaisilla palvelimilla, kuten myös insinöörityön projektien kehitystyökalut. Tämä johtuu siitä, että erilaiset Unixit ovat turvallisia ja vakaita vaihtoehtoja, vaikkakaan eivät käyttäjäystävällisiä.

Yhden yksittäisen ohjelman kanssa ei ollut niinkään vaikeuksia, vaan ongelmallista oli saada useampi kehitys- ja testausympäristön ohjelmisto toimimaan yhdessä. Maven oli ohjelmista alunperin kaikkein tuntemattomin, joten sen toimintakuntoon saamisen kanssa kului verrattain paljon aikaa. Mavenia voisi tutkia lisää, sen kohdalla automatisoinnin prosessissa olisi varmasti virtaviivaistamismahdollisuuksia. Esimerkiksi enemmän voisi tutkia sitä, mikä on Mavenissa todella järkevin tapa määritellä projektin ominaisuudet.

Yksikkötestauksen toteuttaminen Piiskalle oli helppoa aivan alkuvaiheen pienen kömpelyyden jälkeen. JUnit-kehystä oli vaivaton käyttää Eclipsen

käyttöliittymässä. Yksikkötestien kirjoittaminen sujui yllättävän nopeasti, vaikka Piiskassa ei paljoa testattavaa ollutkaan. Toki yksikkötestauksessa riittää kehittämistä ja JUnitin käyttämisessä opettelemista, ennen kuin sen hallitsee hyvin. Esimerkiksi liitteessä 2 näkyvässä testEntry-testimetodissa tarkistuksen ensimmäistä vapaaehtoista parametria ei aluksi käytetty, sillä testaaja ei ollut vielä siinä vaiheessa täysin ymmärtänyt sen olemassaolon hyötyä.

Lisätutkimusta prosessimalleista yleensä ansaitsisi suomalaiskehitteinen Pacing Development -prosessimalli (*PD*) [Rau04]. PD-mallia silmäiltiin tätä insinöörityötä varten, mutta se päätettiin kuitenkin rajata työn ulkopuolelle, sillä kyseessä on enemmän yleisempi projektinhallintaan kehitetty prosessimalli ja PD-mallin periaatteet testauksen suhteen ovat samankaltaisia kuin XP:ssä. Fathammer Oy:n käyttöön PD voisi olla oivallinen, sillä työntekijöitä tarvitsisi suojella raivoisilta hallinnoinnin puuskilta, jotka näyttävät kuuluvan yrityskulttuuriin tällä hetkellä.

Lähteet

AnS03 Anderson, D. ja Schragenheim, E., *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*. Prentice Hall, 2003.

Ant05 Ant-työkalun sivusto, 2005. <http://ant.apache.org/>. Luettu 21.1.2005.

Apa04 Apache HTTP-palvelimen sivusto, 2004. <http://httpd.apache.org/>. Luettu 20.1.2005.

Ast03 Astels, D., *Test-Driven Development: A Practical Guide*. Prentice Hall, 2003.

Bai03 Baird, S., *Sams Teach Yourself Extreme Programming in 24 Hours*. Sams Publishing, 2003.

Bec00 Beck, K., *Extreme Programming Explained*. Addison Wesley, 2000.

Bec02 Beck, K., *Test-Driven Development By Example*. Addison Wesley 2002.

BoT03 Boehm, B. ja Turner, R., *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley, 2003.

BuC03 Burke, E. ja Coyner, B., *Java Extreme Programming Cookbook*. O'Reilly, 2003.

Bur03 Burnstein, I., *Practical Software Testing: a process-oriented approach*. Springer-Verlag, 2003.

CrH03 Crispin, L. ja House, T., *Testing Extreme Programming*. Pearson Education, 2003.

Ecl05 Eclipse-sovelluskehitysympäristön sivut, 2005. <http://eclipse.org/>. Luettu 19.1.2005.

Fat05 Fathammer Oy:n markkinointimateriaali, 2005.

Fre05 FreeBSD Handbook. The FreeBSD Documentation Project, 2005. <http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook>. Luettu 8.2.2005.

Ham95 Hamilton, M., *Software Development: Building Reliable Systems*. Prentice Hall, 1999.

HaM02 Haikala, I. ja Märijärvi, J., *Ohjelmistotuotanto*. 8. uudistettu painos. Talentum Media, 2002.

Hig04 Hightower, R. et al., *Professional Java Tools for Extreme Programming*. Wiley Publishing, 2004.

IES98a *IEEE Standard for Software Test Documentation 892-1998*. The Institute of Electrical and Electronics Engineers, 1998.

IES98b *IEEE Standard for Software Project Management Plans 1058-1998*. The Institute of Electrical and Electronics Engineers, 1998.

Jef04 Jeffries, R., Where's the Spec, the Big Picture, the Design? XPMagazine, 2004. <http://www.xprogramming.com/xpmag/docBigPictureAndSpec.htm>. Luettu 16.2.2005.

JoE94 Jorgensen, P. ja Erickson, C., Object-Oriented Integration Testing. Communications of the ACM, Vol. 37, No. 9, 1994.

JPr04 *JProbe Developer's Guide*. Quest Software, 2004.

Jun05 JUnit-kehyksen sivusto, 2005. <http://junit.org/>. Luettu 18.1.2005.

Mar03 Marchesi, M. et al, *Extreme Programming Perspectives*. Pearson Education, 2003.

Mav05 Maven-työkalun sivusto, 2005. <http://maven.apache.org/>. Luettu 21.1.2005.

McB02 McBreen, P., *Questioning Extreme Programming*. Addison Wesley, 2002.

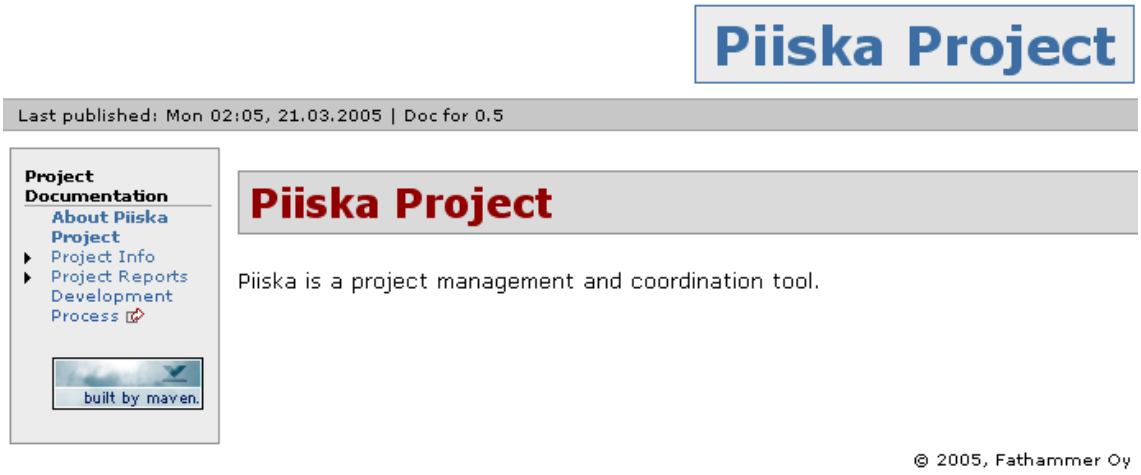
NLL05 Nieminen, M., Project Manager, Fathammer Oy; Lehti, S., Lead Software Engineer, Fathammer Oy; Lauha, J., Senior Software Engineer, Fathammer Oy. Keskustelu projektipäällikön ja muiden projektiryhmän jäsenten kanssa 2.2.2005.

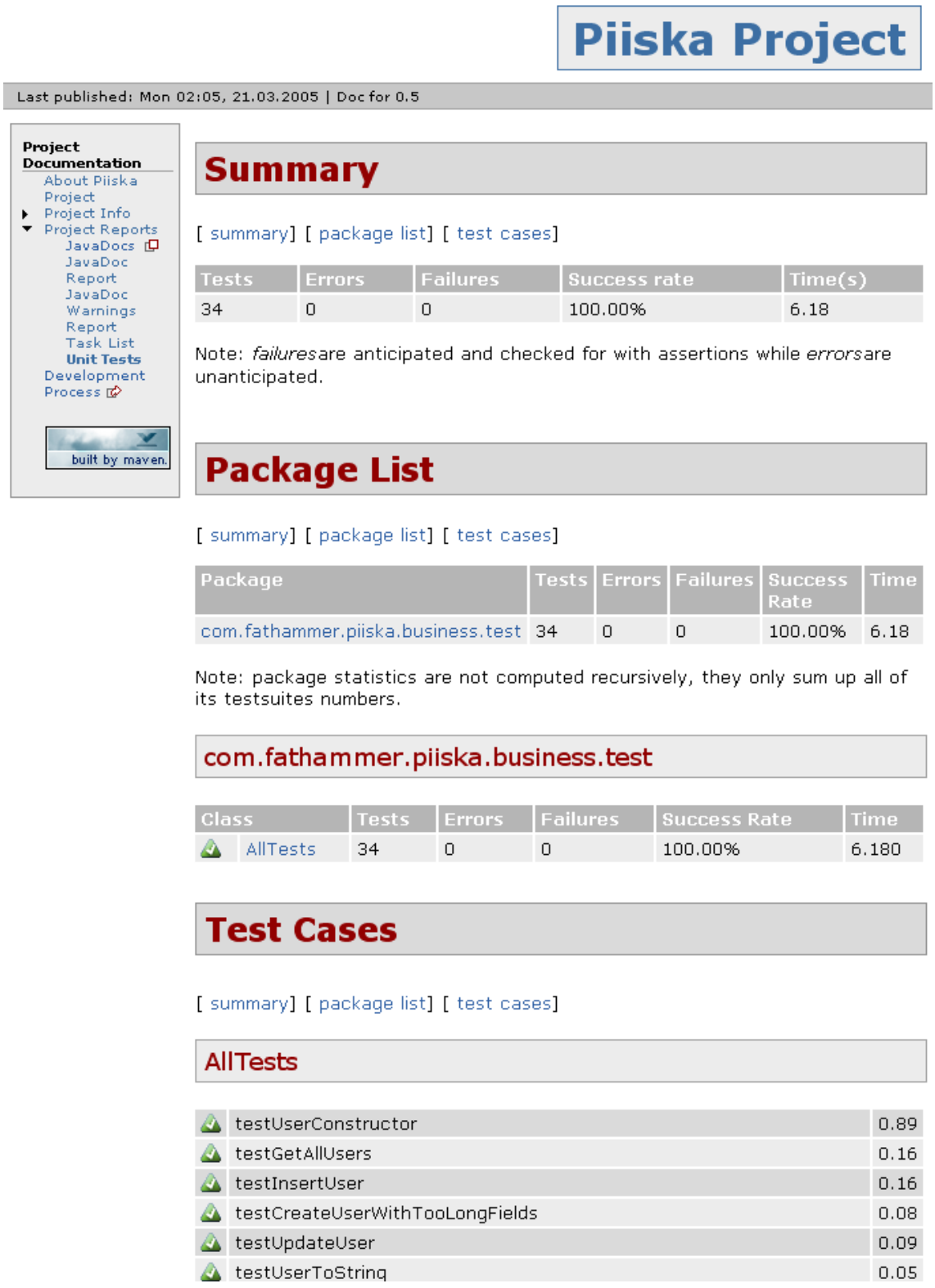
OCW01 OpenCourseWare Lecture Notes, Case Study: JUnit. Massachusetts Institute of Technology, 2001. <http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-170Laboratory-in-Software-EngineeringFall2001/735992D6-3051-4B10-B3F9-30603975224A/0/lecture17.pdf>. Luettu 24.1.2005.

Rau04 Rautiainen, K. et al., *Pacing Software Development: A Framework and Practical Implementation Guidelines*. Otamedia, 2004.

Rie96 Riel, A., *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

Ver05 Verwÿnen, J., Production Director, Fathammer Oy. Keskustelu 24.1.2005.


Piiska Project
Last published: Mon 02:05, 21.03.2005 | Doc for 0.5
Project Documentation
About Piiska Project
Project Info
Project Reports
Development Process
built by maven.
Piiska Project
Piiska is a project management and coordination tool.
© 2005, Fathammer Oy



Piiska Project

Last published: Mon 02:05, 21.03.2005 | Doc for 0.5

Project Documentation
- About Piiska Project
- Project Info
- Project Reports
  - JavaDocs
  - JavaDoc Report
  - JavaDoc Warnings Report
  - Task List
  - **Unit Tests**
- Development Process

built by maven.

Summary

[summary] [package list] [test cases]

Tests	Errors	Failures	Success rate	Time(s)
34	0	0	100.00%	6.18

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Package List

[summary] [package list] [test cases]

Package	Tests	Errors	Failures	Success Rate	Time
com.fathammer.piiska.business.test	34	0	0	100.00%	6.18

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

com.fathammer.piiska.business.test

Class	Tests	Errors	Failures	Success Rate	Time
AllTests	34	0	0	100.00%	6.180

Test Cases

[summary] [package list] [test cases]

AllTests

testUserConstructor	0.89
testGetAllUsers	0.16
testInsertUser	0.16
testCreateUserWithTooLongFields	0.08
testUpdateUser	0.09
testUserToString	0.05

Jatkuu seuraavalla sivulla.

testUserToStringWithNickname	0.05
testLoginUser	0.06
testDeleteUser	0.08
testProjectConstructor	0.09
testGetRootProjects	0.06
testGetAllProjects	0.11
testInsertProject	0.08
testGetSubProjects	0.20
testGetAllSubProjects	0.35
testUpdateProject	0.12
testUpdateParentVectors	0.10
testDeleteProject	0.08
testEntry	0.20
testGetAllEntries	0.19
testGetAllUsersEntriesLong	0.43
testGetUsersEntries	0.22
testGetAllUsersEntriesLongLong	0.29
testGetProjectsEntries	0.13
testInsertEntry	0.14
testUpdateEntry	0.19
testDeleteEntry	0.15
testTaskConstructorLong	0.16
testInsertTask	0.14
testUpdateTask	0.41
testDeleteTask	0.17
testGetTaskLong	0.16
testGetTaskLongDate	0.16
testGetAllTasks	0.18

Luokka TestEntry

```
package com.fathammer.piiska.business.test;

import com.fathammer.piiska.business.Entry;
import com.fathammer.piiska.business.Project;

import junit.framework.TestCase;

public class TestEntry extends TestCase {

     long projectId = 0;
     long entryId;
     String description = "Description for Entry";
     int amount = 8;


     public TestEntry(String nameOfTestMethod) {
          super(nameOfTestMethod);
     }

     protected void setUp() throws Exception {
          Project p = new Project();
          p.insert();
          this.projectId = p.getId();
     }

     protected void tearDown() throws Exception {
          Project p = new Project(projectId);
          p.delete();
     }

     public void testEntry() {
          Entry e = new Entry();
          e.setAmount(amount);
          e.setDescription(description);
          e.setProjectId(projectId);
          e.insert();
          entryId = e.getId();

          try {
               e = new Entry(entryId);
          } catch (Exception exc) {
               exc.printStackTrace();
          }

          String d = ”Entry just made should not be null”;
          assertNotNull(d, e);

          e.delete();
     }
```

```
	public void testUpdateEntry() {
		Entry e = new Entry();
		e.setAmount(amount);
		e.setDescription(description);
		e.setProjectId(projectId);
		e.insert();
		entryId = e.getId();

		e = new Entry(entryId);
		String newDescription = "New description";
		int newAmount = 5;
		e.setDescription(newDescription);
		e.setAmount(newAmount);
		e.update();

		e = new Entry(entryId);
		String d = ”Data should be same after update()”;
		assertEquals(d, newAmount, e.getAmount());
		assertEquals(newDescription, e.getDescription());

		e.delete();
	}
}
```

Luokka AllTests

```
package com.fathammer.piiska.business.test;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

	public static Test suite() {
		TestSuite suite = new TestSuite(
		"Tests for com.fathammer.piiska.business.test");
		//$JUnit-BEGIN$
		suite.addTestSuite(TestUser.class);
		suite.addTestSuite(TestProject.class);
		suite.addTestSuite(TestEntry.class);
		suite.addTestSuite(TestTask.class);
		//$JUnit-END$
		return suite;
	}
}
```

Piiska Test Plan

Change history

Version	Date	Updater	Comment
0.1	20.1.2005	Sara Kapli	Document created
0.2	12.2.2005	Sara Kapli	Updates

Table of Contents

1 Overview

This is a system test plan for project management and working hour tracking software Piiska. Document defines test process in detail, test tools and environment etc.

2 Test Plan Identifier (not needed)

3 Test Items

Piiska

4 Features to Be Tested

Core business logic: When Piiska's interaction with database is tested, the underlying classes for handling the database and Piiska's proper functionality are also tested, regardless of the implementation of the UI layer.

5 Features Not to Be Tested

User interface

6 Approach

Unit tests are main practice of testing this software. Test writer and developer are in most cases different person.

Unit tests are written for every public or protected method in classes to be tested, excluding basic accessors. Unit tests shall also cover constructors that can fail.

Unit test classes are organized into the packages named test. Test package extends the package it tests. Every test package includes test classes named like classes to be tested but with prefix Test. Additionally there is test suite in each test package named AllTests. This suite wraps up all tests in that particular package. *See Figure 1*.

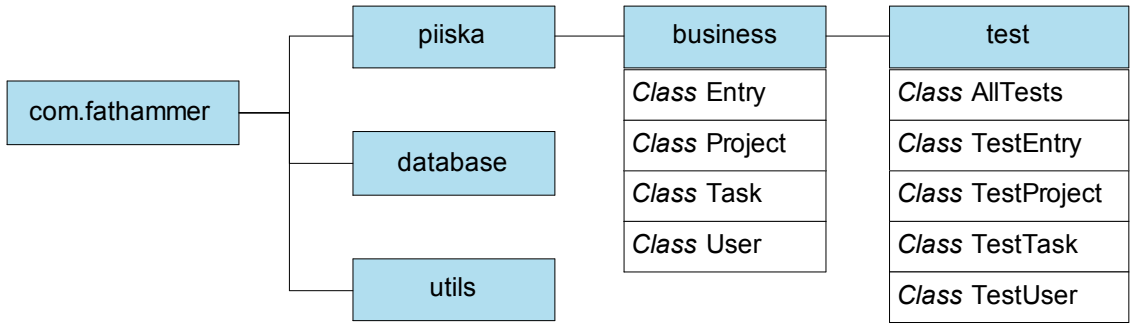


Figure 1 Package structure

7 Items Pass/Fail Criteria

All test items must pass unit tests.

8 Suspension Criteria and Resumption Requirements (not needed)

9 Test Deliverables

- Test plan (this document)
- Test Specifications
- Test input and output data

10 Testing Tasks

See Attachment 1: Task List

11 Environmental Needs

Unit/regression testing is done at the development environment. Initial unit testing will be done at developer's own workstation. Request Tracking -software is used to issue bugs and major test failures for certain developer. Automated nightly build will be triggered by timing program Cron. Development environment consists of one development server, CVS repository server, request tracker server and various workstations. Operating systems and software running on each machine is listed below. Finally Piiska will be moved to appropriate production server and the database to a separate dedicated server. The network between servers and workstations is 100 Mb Ethernet.

- Workstations (Windows XP)
  - Eclipse for Java development

    - JUnit
    - CVS
- Development Server LSD (FreeBSD)
    - Nightly build
    - Nightly unit / regression testing
    - Maven
    - Ant
    - JUnit
    - MySQL database
    - Apache
    - Cron
- CVS Server MOUKARI
    - Central CVS repository
- Request Tracker Server RT
    - Request Tracker

12 Responsibilities

- Sara Kapli
    - Test and build processes
    - Preparing, executing and resolving tests
- Mikko Nieminen
    - Main coding and project managing
    - Managing, checking and resolving tests
- Jyrki Ahpola
    - Technical Support
    - Providing environmental needs

13 Staffing and Training Needs (not needed)

14 Schedule and test durations

Test schedule follows production schedule of Piiska. Duration of automated nightly build is roughly 1 minute and takes place every night 2.05 am. Running tests solely takes about 15 seconds.

15 Risks and Contingencies

- Server hardware malfunction or failure
    - Tests are performed on workstations where applicable

16 Approvals

17 Further Discussion

Some tools, like certain Maven plug-ins, suppose that arrangement of the test classes and packages is different so that test classes reside in respective packages under test tree. For example tool's default would be
src/test/com/fathammer/piiska/business
instead of designed
src/com/fathammer/piiska/business/test.
This is not a critical difference in design, but in some cases it can cause some extra thinking when using such tools. Changing design in following projects would be arguable if there is no need or will to make own plug-in for Maven. On the other hand, writing plug-in for Maven is quite copy-paste. Own plug-in also wraps different sets of goals neatly. Plug-in most probably needs to be written anyways if there is any need to modify intervening goals of the default plug-ins. It could be good practice also if there are some project specific settings like runtime system properties, because plug-in properties are last in process order.

18 Attachment 1: Task List

Task	Prereq.	Special skills	Responsibility	

X-Forge Online Test Plan

Table of Contents

1 Overview

This is a test plan for X-Forge Online, client-server-software for gaming interaction. Document defines test process in detail, test tools and environment etc.

2 Test Plan Identifier (not needed)

3 Test Items

Online

4 Features to Be Tested

Online, follows the specifications for XFO

5 Features Not to Be Tested

6 Approach

Unit tests are main practice of testing this software. Tester and developer are usually different persons.

Unit tests are written for every public or protected method in classes to be tested, excluding basic accessors. Unit tests shall also cover constructors that can fail.

Unit test classes are organized into the packages named test. Test package extends the package it tests. Every test package includes test classes named like classes to be tested but with prefix Test. Additionally there is test suite in each test package named AllTests. This suite wraps up all tests in that particular package.

7 Items Pass/Fail Criteria

Items must pass unit tests in a first phase, integration tests and load tests sufficiently later on.

8 Suspension Criteria and Resumption Requirements (not needed)

9 Test Deliverables

- Test Plan (this document)
- X-Forge Online Test Specifications
- Test input and output data

10 Testing Tasks

11 Environmental Needs

Unit testing and load testing are done at the development environment. Initial unit testing will be done at each developer's own workstation. Unit tests will be automated with Maven and run nightly. Development environment in the beginning of the project consists of three low level servers and various workstations. Operating systems and software running on each server is listed below. Server-side hardware will probably be upgraded to real server hardware and one operating system. The network between servers will be upgraded to gigabit Ethernet in the future.

- Workstations (Windows XP)
- Eclipse for Java development
- JUnit
- CVS
- Server 1 (FreeBSD)
- Master Server (main entry point to X-Forge Online from client point of view)
- Nightly build
- Nightly unit / regression testing
- Server 2 (Debian)
- User Profile Service
- Other services (to be defined later)
- Server 3 (Windows XP)
- MySQL database
- Online Prototype

12 Responsibilities

- Sara Kapli
  - Test and build processes
  - Preparing, executing and resolving tests
- Mikko Nieminen
  - X-Forge Online architecture, server-side implementation and web interfaces
  - Managing, checking and resolving tests
- Samuli Lehti
  - X-Forge Online architecture, client-side implementation
  - Checking tests
- Petrus Lundqvist
  - X-Forge Online inner customer
  - Witnessing tests, providing environmental needs
- Jyrki Ahpola
  - Technical Support
  - Providing environmental needs

13 Staffing and Training Needs (not needed)

14 Schedule and test durations

Overall schedule for testing follows roughly the schedule of XFO. There are yet no estimations of durations of individual tests.

15 Risks and Contingencies

- Server hardware malfunction or failure
  - Tests are performed on workstations where applicable

16 Approvals

