Topi Kasari

Jatkuvan integroinnin koontityökalut

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Tietotekniikka
Insinöörityö
22.4.2012

Metropolia

Tiivistelmä

Tekijä Otsikko Sivumäärä Aika	Topi Kasari Jatkuvan integroinnin koontityökalut 67 sivua + 2 liitettä 22.4.2012
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaajat	Yliopettaja Erja Nikunen Yliopettaja Auvo Häkkinen

Metropolia Ammattikorkeakoulun ohjelmistotuotantoprojekteissa käytetään jatkuvaa integrointia ja iteratiivisia kehitysmenetelmiä. Kehitysympäristö on rakennettu suosituimpien Java-ympäristön koontityökalujen, Antin ja Mavenin, ympärille. Sekä Ant että Maven ovat suosiostaan huolimatta saaneet osakseen paljon kritiikkiä. Mavenia pidetään yleisesti liian jäykkänä työkaluna. Antin ongelma on sen prosessikuvauskielen heikko ilmaisuvoima, joka johtaa helposti ylläpidettävyysongelmiin. Antin ja Mavenin ongelmille on yhteistä niiden prosessikuvauskielien XML-pohjaisuus. XML-kieltä pidettiin Antin ja Mavenin suunnittelun aikaan hyvänä kielenä koontityökaluille. Kokemus on kuitenkin osoittanut, että XML:n ilmaisuvoima ja hierarkkinen rakenne rajoittavat koonnin kuvausta.

Tämän opinnäytetyön ensisijainen tavoite oli tutkia uusia koontityökaluja ja kartoittaa niiden soveltuvuutta Metropolian ohjelmistoprojekteihin. Uusien työkalujen kehittäjät ovat ottaneet Antiin ja Maveniin kohdistuneen kritiikin tosissaan. Suurin osa uusista koontityökaluista hyödyntää Turing-täydelliseen ohjelmointikieleen perustuvaa sovellusaluekohtaista kieltä XML:n sijaan. Buildr ja Gradle ovat esimerkkejä työkaluista, jotka pyrkivät yhdistämään Antin ja Mavenin parhaat puolet. Tutkittavat koontityökalut valittiin niiden suosion mukaan. Valinnoissa otettiin huomioon myös tuki Jenkins CI -koontipalvelimelle, jota käytetään Metropolian kehitysympäristössä. Työkaluja tutkittiin ensin yleisluontoisesti ja muutamia niistä tarkasteltiin tarkemmin PHP- ja Android-sovellusten koonnissa. Lisäksi työssä tutkittiin tietovarastonhallintasovelluksia. Integroidun riippuvuuksienhallinnan sisältävät koontityökalut hyötyvät monissa tapauksissa omasta tietovarastopalvelimesta.

Työssä havaittiin monien vaihtoehtoisten koontityökalujen olevan erittäin varteenotettavia vaihtoehtoja erityisesti Java-projektien koontiin. Gradle on hyvin vakaa ja monipuolinen työkalu, jolla on potentiaalia korvata Ant ja Maven. PHP- ja Android-koonnissa uudet koontityökalut eivät tarjonneet aivan odotetunlaisia etuja verrattuna Antiin. Uusien koontityökalujen ympärillä ei ole yhtä suurta liitännäisekosysteemiä kuin Antilla ja Mavenilla. Tietovarastonhallintasovellukset osoittautuivat mielenkiintoisiksi, ja Metropolian kehitysympäristöä saatetaan tulevaisuudessa laajentaa tietovarastopalvelimella.

Avainsanat	jatkuva integrointi, koonti, riippuvuuksienhallinta, Ant, Maven, Gradle

Metropolia

Abstract

Author Title Number of Pages Date	Topi Kasari Continuous Integration Build Tools 67 pages + 2 appendices 22 April 2012
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software Engineering
Instructors	Erja Nikunen, Principal Lecturer Auvo Häkkinen, Principal Lecturer

The software projects in Metropolia University of Applied Sciences utilize continuous integration and iterative development methods. The development environment was built around the most prominent Java build tools, Ant and Maven. While popular, both of these build tools have been subjected to criticism from the developer community. Maven is often described as overly rigid. Ant suffers from lack of expressiveness that renders complex builds unmaintainable. Many of the issues in both tools stem from their XML based configuration that was once thought to be an end-all solution for build tools. Experience has shown that XML based project definitions are too inexpressive and restrictive for many software builds.

The main objective of this thesis was to research new build tools that are rising in popularity and assess their suitability for the projects done in Metropolia. The developers of these new build tools have taken the criticism of Ant and Maven seriously. Vast majority of the alternative tools utilize a domain-specific language based on a Turing complete programming language instead of XML. Tools such as Buildr and Gradle attempt to combine the best parts of Ant and Maven while adding in their own flavor. The researched tools were chosen based on their popularity and support in Jenkins CI, the continuous integration server of choice in the development environment. The tools were first analyzed from a general perspective. After initial evaluation some of the tools were used in two case studies in building PHP and Android software. As a secondary objective, a study was carried out on repository managers, which can further enhance the capabilities of modern build tools that integrate dependency management into the build automation process.

The research showed that many of the alternative build tools delivered on their promises. Especially Gradle is a very mature and complete tool that has the potential to replace Ant and Maven. The case studies, however, showed that Ant is still a strong contender in many use cases. The new build tools cannot match the large plugin ecosystems that Ant and Maven boast. The study into repository managers proved fruitful and a repository management server may be added to the environment in the future.

Keywords	continuous integration, building, dependency management, Ant, Maven, Gradle

Sisällys

Lyhenteitä ja määritelmiä

Agilefant	Projektinhallintasovellus ketterän ohjelmistotuotannon tarpeisiin.
Android	Open Handset Alliancen kehittämä mobiilikäyttöjärjestelmä.
Ant	Apache Software Foundationin kehittämä suosittu koontityökalu. Katso luku 2.2.
Archiva	Apache Software Foundationin kehittämä tietovarastonhallintasovellus.
Artifactory	JFrog Ltd:n kehittämä tietovarastonhallintasovellus. Katso luku 4.1.
Buildr	Apache Software Foundationin kehittämä koontityökalu. Katso luku 3.2.
Emma	Testikattavuustyökalu Javalle.
Gant	Koontityökalu, joka käyttää ytimenään Antia. Katso luku 3.4.
Gradle	Gradlewaren kehittämä koontityökalu. Katso luku 3.5
Hudson	Sonatypen kehittämä koontipalvelinsovellus.
Integroitu kehitysympäristö	*Integrated Development Environment*. IDE. Kaikki keskeiset sovelluskehitykseen liittyvät työkalut integroiva sovellus.
Ivy	Apache Software Foundationin kehittämä riippuvuuksienhallintatyökalu, jota käytetään usein yhdessä Antin kanssa.
Jatkuva integrointi	Prosessi, jossa sovellusta käännetään, testataan ja integroidaan jatkuvasti kehityksen aikana.

Jenkins CI	Suosittu koontipalvelinsovellus.
Kehitysversio	*Snapshot*. Koontituote joka on aktiivisen kehityksen alainen. Version päätteeksi merkataan SNAPSHOT.
Ketterä ohjelmistokehitys	*Agile software development*. Kokoelma moderneja ohjelmistotuotantomenetelmiä, jotka pyrkivät virtaviivaistamaan sovelluskehitystä.
Koontituote	*Build artifact*. Koonnin välituote, esimerkiksi ajettava binääri, kirjasto tai dokumentaatio.
Koontityökalu	Koontiprosessin automatisoiva työkalu.
Koontipalvelin	Palvelin, jolla koonti suoritetaan. Koontipalvelimella saatetaan tarkoittaa myös jatkuvan integroinnin koontipalvelinsovellusta (esim. Jenkins CI), jota koontipalvelimella ajetaan.
Make	Yksi vanhimmista ja suosituimmista koontityökaluista.
Maven	Apache Software Foundationin kehittämä koontityökalu. Katso luku 2.2.
Moduulipalvelin	Katso tietovarastopalvelin.
Nexus	Sonatypen kehittämä tietovarastonhallintasovellus. Katso luku 4.2.
Phing	PHP-koontiin erikoistunut koontityökalu. Katso luku 3.6.
Projektioliomalli	Mavenin prosessikuvaus, tyypillisesti nimetty pom.xml:ksi.
Prosessikuvaus	Koontityökalulle syötettävä kuvaus, joka määrittelee koonnin vaiheet ja muut tekijät. Prosessikuvauksen sisältävää tiedostoa voi kutsua koontitiedostoksi.

Rake	Make-sukuinen koontityökalu, joka käyttää Ruby-pohjaista sovellusaluekohtaista kieltä. Katso luku 3.1.
Sbt	Simple Build Tool. Scala-projektien koontiin erikoistunut koontityökalu.
SCons	Koontityökalu, joka on suosittu erityisesti C/C++/Fortran-projekteissa. Katso luku 3.3.
Scrum	Ketterässä kehityksessä käytetty projektinhallintamenetelmä.
Sovellusaluekohtainen kieli	*Domain-Specific Language*. Jonkin sovellusalueen ongelmanratkaisuun tarkoitettu ohjelmointikieli. Monet koontityökalut hyödyntävät yleiskäyttöisen ohjelmointikielen päälle rakennettua sovellusaluekohtaista kieltä.
Subversion	Suosittu versionhallintasovellus.
Tietovarastopalvelin	Tietovarastonhallintasovellusta pyörittävä palvelin. Sovelluksia ovat esim. Artifactory ja Nexus. Katso luku 4.
Turing-täydellinen	Ohjelmointikielen ominaisuutena Turing-täydellisyydellä tarkoitetaan sitä, että kielellä voi simuloida mitä tahansa tietokonetta.
Versionhallintapalvelin	Versionhallintasovellusta, kuten Subversionia, ajava palvelin.

1 Johdanto

Koontityökalut ovat olleet keskeisessä osassa ohjelmistokehityksen työkaluhierarkiassa jo pitkään. Erityisen tärkeäksi niiden rooli on kasvanut ketterien, iteratiivisten sovelluskehitysmenetelmien myötä. Ketterässä sovelluskehityksessä käytetään usein tuottavuutta ja laatua parantavia menetelmiä, kuten jatkuvaa integrointia ja automatisoitua testausta. Automatisointi on helpointa toteuttaa älykkäällä koontityökalulla, jolla koonnin vaiheet voidaan suorittaa. Jatkuvaan integrointiin käytetään tyypillisesti koontipalvelinta, joka ajastaa koonnin ja tarjoaa kehittäjille palautetta.

Metropolian ohjelmistotekniikan opiskelijat osallistuvat opintojensa loppuvaiheessa ohjelmistotuotantoprojektikurssille. Kurssilla kehitetään koululle tai ulkoiselle asiakkaalle sovellus, jonka tekemisessä hyödynnetään ammattimaisia sovelluskehitysmenetelmiä ja työkaluja. Projektinhallintaan käytetään Scrumia ja ketterää kehitystä johdetaan jatkuvan integroinnin avulla. Projekteja varten on perustettu kehitysympäristö, jossa hyödynnetään suosituimpia Java-maailman koontityökaluja, Apache Antia ja Apache Mavenia. Ympäristöä ja siihen liittyviä työkaluja kuvataan tarkemmin luvussa 2.

Ant ja Maven ovat saaneet osakseen paljon kritiikkiä erinäisistä syistä, joita käsitellään myös toisessa luvussa. Viime vuosina koontityökalumarkkinoille on saapunut lukuisia uusia haastajia, jotka pyrkivät korjaamaan niissä havaittuja ongelmia. Muutamat näistä työkaluista ovat saaneet suosiota kehittäjäyhteisöissä ja korkean profiilin avoimen lähdekoodin projekteissa. Tässä opinnäytetyössä kartoitetaan tällaisten tuntemattomampien, nousussa olevien koontityökalujen soveltuvuutta ohjelmistotuotantoprojektien jatkuvan integroinnin ympäristöön. Kiinnostavia koontityökaluja ovat muun muassa Gradle, Buildr ja Phing. Koontityökaluja evaluoidaan luvussa 3.

Nykyisin ohjelmistoprojekteilla on lähes aina riippuvuuksia ulkoisista komponenteista. Näiden komponenttien tuomista projektin käytettäväksi kutsutaan riippuvuuksienhallinnaksi. Nykyiset koontityökalut ja erikoistuneet riippuvuuksienhallintatyökalut pystyvät kätevästi hallitsemaan riippuvuuksia lataamalla määritellyt komponentit julkisista tietovarastoista. Ongelmaksi saattaa kuitenkin muodostua riippuvuuksien saatavuus erityisesti pitkällä tähtäimellä. Usein yrityksissä hyödynnetäänkin tietovarastosovelluksia, jotka toimivat koontityökalun ja julkisen tietovaraston välissä. Tällaiseen tietovarastoon

voidaan julkaista myös yrityksen sisäiset kirjastot ja projektit, joita pystytään tämän jälkeen kätevästi hyödyntämään muissa projekteissa. Metropolian ohjelmistoprojektit saattaisivat hyötyä moduulipalvelimesta, joka ajaisi tietovarastonhallintasovellusta. Tätä mahdollisuutta tutkitaan luvussa neljä.

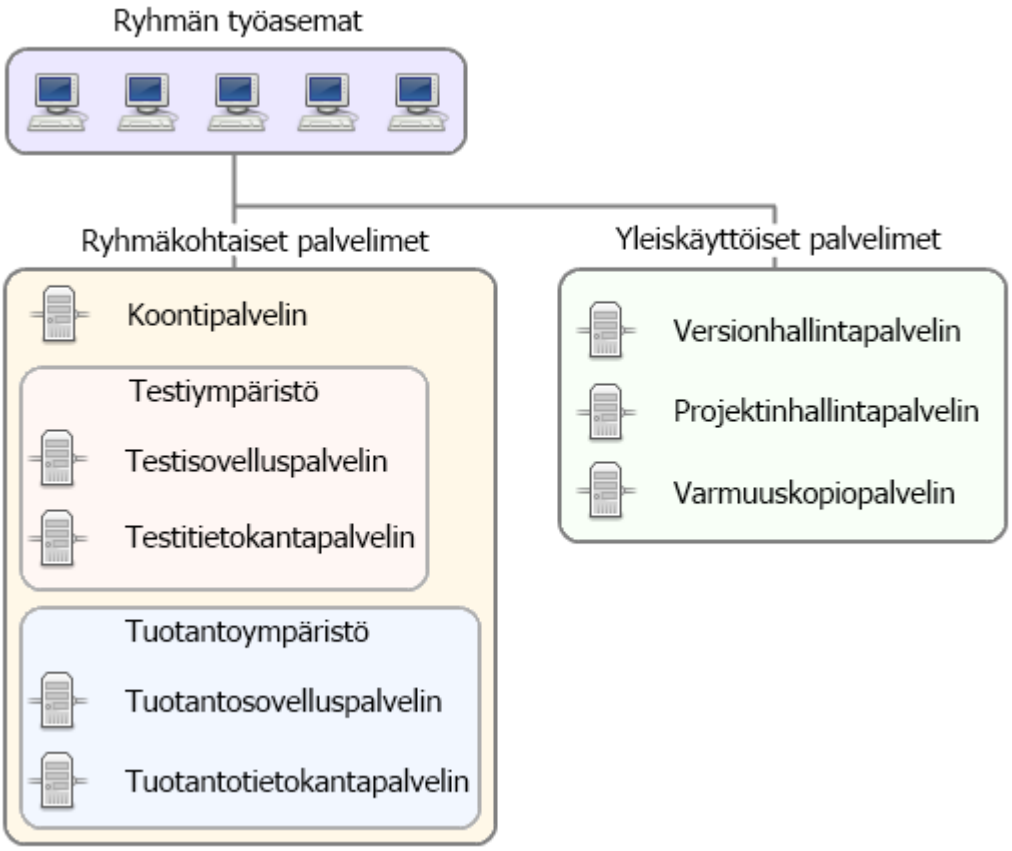
Useimmat Metropolian ohjelmistoprojektit ovat web-sovelluksia. Java-pohjaisten projektien koonti onnistuu nykyisessä ympäristössä hyvin. Osa projekteista toteutetaan PHP:llä, eikä ympäristö nykyisellään tarjoa kunnollista ratkaisua PHP-projektien jatkuvaan integrointiin. Kurssilla on kehitetty myös useampia Android-sovelluksia, joille ympäristössä ei toistaiseksi ole tukea. Viides ja kuudes luku keskittyvät PHP- ja Android-projektien koontiin ja hallitsemiseen käytännössä. Koontityökalut valitaan kolmannen luvun johtopäätöksien perusteella. Projektien riippuvuuksienhallinnassa hyödynnetään neljännen luvun päätelmiä ja mahdollista moduulipalvelintoteutusta.

2 Ohjelmistotuotantoprojektin kehitysympäristö

Metropolian ohjelmistotuotantoprojekteja varten on perustettu kehitysympäristö, joka koostuu virtuaalipalvelimista ja opiskelijoiden käyttämistä työasemista. Ympäristö on rakennettu jatkuvan integroinnin mahdollisimman sujuvaa hyödyntämistä varten. Ryhmillä on käytössään useita suoritusympäristöjä, joita voidaan käyttää sovelluksen testaamiseen ja tuotantoympäristön simuloimiseen. Tässä luvussa kuvataan ympäristön rakennetta, eri palvelimien tarjoamia palveluita ja erityisesti ympäristössä hyödynnettäviä koontityökaluja.

2.1 Kehitysympäristön yleiskuvaus

Jokaisella ryhmällä on kehitysympäristössä viisi työasemaa, joilla varsinaista kehitystyötä tehdään. Lisäksi ryhmäläisillä on käytössään viisi ryhmäkohtaista palvelinta. Ryhmäkohtaisista palvelimista keskeisin on koontipalvelin, jolla ajetaan Jenkins CI -sovellusta. Koontipalvelimen toimintaa kuvataan tarkemmin luvussa 2.3. Kehitettävät sovellukset ovat usein web-sovelluksia, minkä vuoksi neljä muuta ryhmäkohtaista palvelinta keskittyvät web-sovelluksien testaamiseen ja ajamiseen. Ryhmäläisillä on käytössään kaksi sovelluspalvelinta, joilla voidaan ajaa esimerkiksi Java- ja PHP-sovelluksia. Toinen palvelimista on tarkoitettu testaukseen ja toinen tuotantoympäristöksi. Vastaavasti kaksi tietokantapalvelinta on käytettävissä yhdessä näiden sovelluspalvelimien kanssa. Kuvassa 1 on esitetty ympäristön rakenne.



Kuva 1. Kehitysympäristön laitteisto

Jatkuvan integroinnin ympäristö on perustettu lähinnä Java-sovelluksien kehitystä ajatellen [Lukkarinen 2011: 1]. Jenkins CI kuitenkin tukee modulaarisen rakenteensa ansiosta lukuisia ohjelmointikieliä ja kehitykseen liittyviä työkaluja. Ympäristöön on siis periaatteessa helppo liittää työkaluja, joiden avulla ympäristö tukisi monia eri ohjelmointikieliä ja -ympäristöjä. Käytännössä tehtävä vaatii työkalujen tuntemista ja parhaiden toimintatapojen arviointia. Koontipalvelimen lisäksi työasemat, sovelluspalvelimet ja tietokantapalvelimet tarvitsevat uusia sovelluksia ja konfiguraatiomuutoksia, kun jatkuvan integroinnin ympäristöön liitetään tuki uudelle sovellusalustalle. Tuki PHP-sovelluksien koontiin on liitetty ympäristöön ympäristön perustamisen jälkeen. Esimerkiksi C++-sovellusten koontiin ei ole ympäristössä tällä hetkellä tukea, koska projekteja ei ole juurikaan kielellä tehty.

Tuotteen versionhallintaan ja projektinhallintaan on kaksi erillistä palvelinta, jotka ovat kaikille ryhmille yhteiset. Versionhallintapalvelimella keskeisin sovellus on Subversion. Kaikilla ryhmillä on käytössään kaksi Subversion-tietovarastoa, joista toista käytetään sovelluskoodin versionhallintaan ja toista oheistuotteiden, kuten dokumentaation talti-

oimiseen. Projektinhallintapalvelimella ajetaan yhtä Agilefant-instanssia jokaista ryhmää varten. Agilefant on projektinhallintatyökalu, joka on kehitetty erityisesti ketterän kehityksen tarpeisiin. Ryhmät käyttävät sitä kehityksen suunnitteluun, Scrumin työlokin ylläpitoon ja tehtyjen työtuntien merkkaamiseen.

2.2 Kehitysympäristön koontityökalut

Useimmat ohjelmistotuotantoprojektit toteutetaan Javalla, joten pääpaino ympäristön kehittämisessä on ollut Java-maailman koontityökaluilla. Java-maailmassa suosituin koontityökalu on pitkään ollut Apache Ant. Se tehtiin korvaamaan Make, joka ei ole alustariippumaton koontityökalu [Apache Ant - Frequently Asked Questions 2011]. Ant on edelleen erittäin suosittu, mutta sen suosiota ovat viime vuosina syöneet uudet tulokkaat. Apache Maven on noussut Antin rinnalle erittäin suosituksi koonti- ja riippuvuuksienhallintatyökaluksi. Sekä Ant että Maven ovat nykyisin hyvin kypsiä työkaluja, joita käytetään monissa yrityksissä. Nämä työkalut ovat myös Metropolian opiskelijoilla tuttuja, mikä onkin tarpeen, koska koontityökalujen käyttö vaatii jonkinasteista asiantuntemusta. Kehitysympäristössä on käytössä sekä Ant että Maven, ja niiden käyttäminen on hyvin dokumentoitu.

Apache Ant

Ant on perinteinen koontityökalu, jonka prosessikuvaustiedostossa kuvataan koonnin vaiheet hyvin yksityiskohtaisesti. Kehittäjällä on täysi vapaus ja vastuu rakentaa koontityö erilaisista vaiheista, joiden riippuvuussuhteet täytyy myös kuvata. Ant ei aseta minkäänlaisia vaatimuksia projektin rakenteelle. Tämän joustavuuden ansiosta Antia voi käyttää yleispätevänä koontityökaluna. Se ei ole sidottu mihinkään tiettyyn ohjelmointialustaan tai teknologiaan. Ant on alun perin kehitetty Java-ohjelmien koontiin, mutta sitä voi helposti hyödyntää myös esimerkiksi C++- ja PHP-projektien koonnissa. Mikään ei myöskään estä käyttämästä Antia yleisenä työkaluna esimerkiksi monimutkaiseen tiedostojen kopiointioperaatioon, jota täytyy suorittaa usein. Ant on melko yksinkertainen koontityökalu, eikä sen omaksumiseen joudu kuluttamaan kohtuuttomasti aikaa. Monet ominaisuusrikkaat koontityökalut vaativat huomattavasti syvempää perehtymistä, jotta niitä voi mukavasti käyttää.

Antin prosessikuvaustiedostot käyttävät XML-pohjaista sovellusaluekohtaista kuvauskieltä. Suuri osa Antiin kohdistuvasta kritiikistä liittyy XML-kieleen. XML on myös tiukasti hierarkkinen, mikä rajoittaa sen ilmaisuvoimaa. Keskeisin ongelma XML:ssä on sen hankala luettavuus ja monisanaisuus. XML on periaatteessa helppo kieli lukea ja ymmärtää, vaikkei sovelluskehityksestä ymmärtäisikään mitään. Käytännössä pitkät XML-tiedostot ovat kuitenkin hankalia sisäistää, koska XML-kielen rakenteet vievät leijonanosan tilasta, ja ihminen kiinnittää niihin huomiota. Luettavuuden ongelmat ovat paljolti vältettävissä käyttämällä XML:n muokkaamiseen ja lukemiseen ohjelmaa, joka sisältää XML-kielen syntaksikorostuksen. Koodiesimerkissä 1 on esitetty minimaalinen prosessikuvaus Java-sovelluksen koontiin. Prosessikuvauksella voidaan kääntää projektin ohjelmakoodi ja paketoida sovellus suoritettavaksi jar-pakkaukseksi.

```
<project name="java-simple" default="package" basedir=".">

    <property name="sourceDir" location="src"/>
    <property name="buildDir" location="build"/>
    <property name="classesDir" location="${buildDir}/classes"/>
    <property name="libsDir" location="${buildDir}/libs"/>

    <target name="clean">
        <delete dir="${buildDir}"/>
    </target>

    <target name="init">
        <mkdir dir="${classesDir}"/>
        <mkdir dir="${libsDir}"/>
    </target>

    <target name="compile" depends="init">
        <javac srcdir="${sourceDir}" destdir="${classesDir}"
          includeantruntime="false"/>
    </target>

    <target name="package" depends="compile">
        <jar jarfile="${libsDir}/${ant.project.name}.jar" basedir="${classesDir}">
            <manifest>
                <attribute name="Main-Class" value="fi.metropolia.esimerkki.Main"/>
            </manifest>
        </jar>
    </target>

</project>
```

Koodiesimerkki 1. Ant-prosessikuvaus Java-sovelluksen koontiin.

Toinen XML:ään liittyvä ongelma on Antin käyttämän sovellusaluekohtaisen kielen imperatiivisuus. Antilla voidaan tehdä paljon asioita deklaratiivisesti, lyhyillä koontia kuvaavilla ilmaisuilla. Antin ydintehtävät (Ant Core Tasks) noudattavatkin ajatusta deklaratiivisesta koonnin kuvauksesta. Tästä huolimatta Antia käytettäessä koetaan usein

tarpeelliseksi hyödyntää ohjauslauseita, kuten ehtolauseita ja silmukoita. Ant Contrib -projekti tarjoaakin liitännäisiä, jotka mahdollistavat monimutkaisemman sovelluslogiikan kirjoittamisen XML:llä. Koodiesimerkissä 2 on esitetty silmukan ja ehtorakenteen käyttämistä Ant Contribin avulla.

```
<project name="silmukka-ehto" default="default">

    <taskdef resource="net/sf/antcontrib/antlib.xml"/>

    <target name="default">
        <for list="1,2,3,4,5,6,7,8,9,10" param="n">
            <sequential>
                <math result="result" operand1="@{n}"
                      operation="%" operand2="2" datatype="int"/>
                <if>
                    <equals arg1="${result}" arg2="0"/>
                    <then>
                        <echo>@{n}</echo>
                    </then>
                </if>
            </sequential>
        </for>
    </target>

</project>
```

Koodiesimerkki 2. Yksinkertainen Ant-prosessikuvaus, joka tulostaa parilliset luvut yhden ja kymmenen väliltä käyttämällä silmukkaa ja ehtorakennetta.

Ohjauslauseiden käyttäminen voi olla käytännöllistä, mutta tästä on helposti seurauksena se, että Antin koontitiedostoihin ilmaantuu paljon XML:llä kirjoitettua sovelluslogiikkaa. XML ei kuitenkaan ole skripti- tai ohjelmointikieli, ja sen syntaksi soveltuu huonosti imperatiiviseen ohjelmointiin [Berglund & McCullough 2011]. Modernit koontityökalut ovat myös siirtyneet tapaan, jossa kokonainen projekti voidaan kuvata deklaratiivisesti. Antilla voi kuvata yksinkertaisia tehtäviä *(target)* deklaratiivisesti, mutta lopulta Ant-projektit koostuvat useista tehtävistä, joiden suoritusjärjestys on eksplisiittisesti prosessikuvauksessa määritelty. [Ant: A Critical Retrospective.]

Ant ei ole riippuvuuksienhallintatyökalu, mutta riippuvuudet voidaan hallita erillisellä työkalulla, joka integroituu Antiin. Ylivoimaisesti suosituin ja standardiksi muodostunut ratkaisu on käyttää Antin kanssa riippuvuuksienhallintaan Apache Ivyä. Ivy on virallinen Antin aliprojekti, mutta sitä voi käyttää myös ilman Antia. Antin ja Ivyn yhteiskäyttö on hyvin suosittu tapa hallita projektin riippuvuuksia menettämättä Antin tarjoamaa joustavuutta koonnissa. Antin ja Ivyn yhteiskäyttö voi kuitenkin olla kankeaa integ-

roidusta kehitysympäristöstä. Riippuvuudet täytyy myös määritellä erillisessä tiedostossa, mikä ei ole kovin kätevää verrattuna esimerkiksi Mavenin projektioliomalliin. Maven integroituu hyvin kaikkiin suosittuihin kehitysympäristöihin, mikä on yksi sen suurimmista valteista. Ivyn käyttöä Antin kanssa ei käsitellä tässä työssä, koska monet luvussa 3 esiteltävistä työkaluista käyttävät Ivya riippuvuuksienhallintaan. Ivya on helpointa käyttää moderneista koontityökaluista, joihin se integroituu saumattomasti.

Riippuvuuksienhallinta on erittäin tärkeä osa Java-projektien kokonaisvaltaista hallintaa. Useat koontityökalut, kuten Maven, tarjoavat valmiiksi integroituna riippuvuuksienhallinnan. Mavenin ja Antin eroavaisuudet eivät kuitenkaan rajoitu riippuvuuksienhallintaan. Maven ja useat muut koontityökalut perustuvat käytäntöihin, joita kehittäjän täytyy noudattaa. Käytännöt tarkoittavat toimintatapoja, joiden mukaisesti koontityökalu toimii, vaikkei kehittäjä eksplisiittisesti niin käskisi. Näistä käytännöistä voidaan usein poiketa, mutta se voi vaatia jopa enemmän työtä kuin Antin yksityiskohtaiset määrittelyt. Ant onkin edelleen suosittu juuri sen joustavuuden vuoksi. Monimutkaiset, tietylle projektille spesifit koonnin vaiheet voivat olla hyvin vaikeita toteuttaa Mavenilla. [Maven: The Complete Reference.]

Apache Maven

Apache Maven on Java-maailman suosituin koontityökalu Antin ohella. Maven edustaa hyvin erilaista lähestymistapaa koonnin kuvaukseen kuin Ant. Maven noudattaa tiettyjä käytäntöjä, jotka käyttäjän täytyy tuntea suunnitellessaan projektin rakennetta ja koontia. Tätä toimintafilosofiaa kutsutaan käytännöksi ennen konfiguraatiota *(convention over configuration)*. Ajatusmallin sisäistäminen on olennaista, jotta Mavenin käyttö on mahdollista edes perustasolla.

Kuten Ant, myös Maven käyttää XML-pohjaista sovellusaluekohtaista kieltä koontitiedostossaan. Mavenin koontitiedostoa kutsutaan projektioliomalliksi *(project object model)*. Tyypillisesti koontitiedosto nimetään pom.xml:ksi. Maven edustaa hyvin erilaista lähestymistapaa koonnin kuvaukseen kuin Ant. Mavenin projektioliomalli on tulospohjainen ja esittelevä kuvaus koonnin vaiheista. Projektit voidaan jakaa useammaksi aliprojektiksi, joilla on kaikilla oma projektioliomallinsa. Mavenin ydin on lähinnä kehys liitännäisille, jotka tarjoavat koonnin päämääriä *(goal)*. Maven suorittaa koonnin vai-

heet niin sanotun elämänkaaren *(lifecycle)* mukaisesti. Koonnin elämänkaari koostuu Mavenissa useista vaiheista *(phase)*.

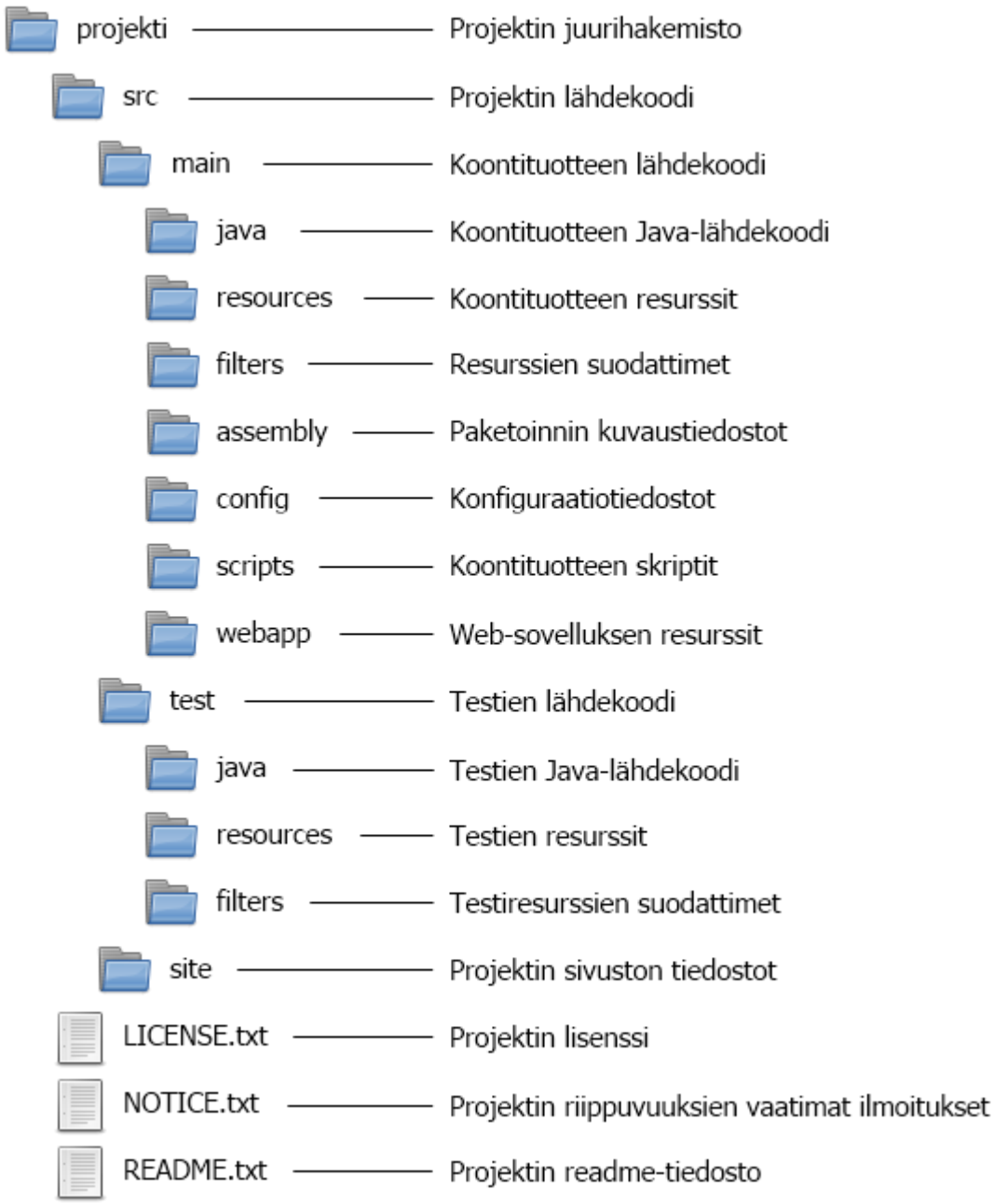
Mavenin projektioliomallin oletusarvot käyvät hyvin Java-projektien koontiin. Yksinkertaisimmillaan koonnin kuvaukseen riittää koodiesimerkin 3 prosessikuvaus.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>fi.metropolia.esimerkki</groupId>
    <artifactId>esimerkki</artifactId>
    <version>1.0.0</version>
</project>
```

Koodiesimerkki 3. Yksinkertainen projektioliomalli

Prosessikuvauksessa määritellään ainoastaan koontituotteen tunnisteita ja projektioliomallin versio (modelVersion). Yhdessä kaikki tunnisteet muodostavat koontituotteen uniikin nimen. Esimerkin prosessikuvauksessa ei määritellä mitään varsinaiseen koontiprosessiin liittyvää, vaan käytetään Mavenin oletusarvoja. Oletusarvoilla Maven pystyy kääntämään Java-lähdekoodin, suorittamaan yksikkötestit ja paketoimaan sovelluksen jar-pakkaukseksi.

Maven on myös onnistunut standardoimaan oletusarvona käyttämänsä hakemistorakenteen, joka on esitetty kuvassa 2.



Kuva 2. Mavenin standardihakemistorakenne [Introduction to the Standard Directory Layout].

Projektin hakemistorakennetta voi muuttaa projektioliomallia konfiguroimalla, mutta oletusrakenteen käyttäminen on hyvin suositeltavaa. Mavenin lisäksi standardirakennetta hyödyntävät myös monet koontityökalut, kuten Buildr, Gradle ja Sbt, joita käsitellään luvussa 3. Rakenteen käyttö on usein suositeltavaa myös muita koontityökaluja käytettäessä, koska se on suurelle osalle kehittäjistä tuttu ja yleisesti hyväksytty.

Standardihakemistorakenne soveltuu hyvin erilaisille projekteille. Tämän vuoksi se saattaa vaikuttaa turhan monimutkaiselta esimerkiksi yksinkertaista Java-projektia varten. Rakenteen käyttämisestä on kuitenkin selviä etuja, koska useat koontityökalut tukevat sitä suoraan ja kehittäjät tuntevat sen. Java-projektissa sovelluksen lähdekoodit sijoitetaan src/main/java-alihakemistoon. Testien lähdekoodit tulevat src/test/java-alihakemistoon. Vastaavasti esimerkiksi Groovylla kehitettäessä sovelluksen lähdekoodit tulisivat hakemistoon src/main/groovy. Standardi määrittelee myös muita hakemistoja ja tiedostoja, joita projektit voivat tarvittaessa hyödyntää. Tavallisesti projektin hakemistorakennetta on hyvä kuvailla readme-tiedostossa.

Luvussa 2 mainitut XML-pohjaisuudesta johtuvat ongelmat Antin sovellusaluekohtaisessa kielessä koskevat suurelta osin myös Mavenia, koska se käyttää XML-pohjaista kieltä. Kuten luvussa todettiin, XML ei ole optimaalinen kieli koontiskriptien tekemiseen. Mavenin projektioliomallin ja koonnin elämänkaaren ansiosta prosessikuvaukset pysyvät usein melko lyhyinä ja luettavina, mikäli Mavenin ja sen liitännäisten tarjoamista käytännöistä ja oletusarvoista ei poiketa. Ongelmiin kuitenkin ajaudutaan nopeasti, kun halutaan tehdä jotain oletuksista poikkeavaa. Satojen rivien projektioliomallit ovat tavallisia, suurissa projekteissa rivimäärät liikkuvat helposti tuhansissa.

Maven-projektien täytyy noudattaa jäykkää vakiomallia, joka ei välttämättä kaikille projekteille ole sovelias. Suurin osa Maveniin kohdistuvasta kritiikistä liittyy jollain tavalla sen jäykkyyteen. Monet suhteellisen yksinkertaiset asiat, kuten tiedostojen kopiointi ja zip-pakkauksen luonti tietyistä tiedostoista, vaativat erilaisia liitännäisiä ja lukemattomia rivejä sovellusaluekohtaista XML:ää. Liitännäisten tekeminen onkin arkipäivää Mavenissa, koska muulla tavoin Mavenin koonnin elämänkaarta ei edes pääse muokkaamaan. Mavenin elämänkaari ei myöskään ole uudelle käyttäjälle helppo ymmärrettävä. Liitännäisten kehittäminen ja niiden liittäminen sopivaan elämänkaaren vaiheeseen vaatii suhteellisen syvällistä Mavenin osaamista. Niin kutsutut seuraavan sukupolven koontityökalut ovat ottaneet jäykkyyteen kohdistuneen kritiikin tosissaan. Luvussa 3 käydään läpi tuoreita koontityökaluja ja huomioidaan erityisesti niiden joustavuus. Huomattavaa on myös, että lähes kaikki uudet koontityökalut käyttävät prosessikuvaukseen johonkin täydelliseen ohjelmointi- tai skriptikieleen perustuvaa sovellusaluekohtaista kieltä. Onkin selvää, että tulevaisuudessa XML ei tule olemaan isossa roolissa koontityökalujen prosessikuvauksia tehtäessä.

2.3 Jenkins CI -koontipalvelin

Jatkuvan integroinnin sydän on koontipalvelin. Metropolian ohjelmistoprojektit käyttävät koontipalvelinsovelluksena Jenkins CI:tä. Jenkins CI tunnettiin aikaisemmin nimellä Hudson, mutta Oraclen ostettua Sun Microsystemsin Hudson siirtyi Oraclen omistukseen. Kehittäjäyhteisö jatkoi Hudsonin kehitystä erillisessä kehityshaarassa uudella Jenkins CI -nimellä keväällä 2011. Oracle jatkaa yhdessä Sonatypen kanssa Hudsonin kehitystä. Hudson ja Jenkins ovat jo kehittyneet jossain määrin erilleen. Hudsonin kehityksessä keskitytään hyvään integraatioon Sonatypen muiden tuotteiden, kuten Mavenin ja Nexuksen kanssa. Liitännäisiä kehittävä yhteisö siirtyi käyttämään Jenkinsiä, joten Jenkins tarjoaa laajemman työkalupaletin käytettäväksi koontipalvelimen kanssa. Hudsonin kehityksen keskittyminen Sonatypen tuotteiden integrointiin vaikuttaa myös tietovarastosovelluksen valintaan luvussa 4. [Bayer 2011; Smart 2011; Van Zyl 2011.]

Jenkins on yksi suosituimmista koontipalvelinsovelluksista, ja modulaarisen rakenteensa ansiosta se soveltuu erittäin hyvin Metropolian ohjelmistotuotantoprojekteihin. Projekteissa käytetään useita ohjelmointikieliä ja tekniikoita, joten koontipalvelimelta tarvitaan mahdollisimman laaja tuki erilaisille kehitysalustoille. Jenkinsin käyttäjäkunta on hyvin laaja ja julkaistuja liitännäisiä on saatavilla useita satoja. Myös Jenkinsin ytimen kehitys on hyvin aktiivista. Jenkinsistä julkaistaan uusi vakaa versio viikoittain.

Jenkins CI täytyy ottaa huomioon myös vaihtoehtoisia koontityökaluja vertailtaessa, sillä se ei tue kaikkia tässä raportissa esille tuotuja koontityökaluja. Tuetut työkalut ovat ehdottomasti kiinnostavampia, koska ne voidaan ottaa helposti käyttöön jatkuvan integroinnin ympäristössä. Jenkinsiin voi suhteellisen vaivattomasti tehdä omia liitännäisiä haluamilleen koontityökaluille, mutta työmäärä on tämän työn puitteissa liian suuri. Lisäksi voidaan olettaa, että tukemattomat koontityökalut eivät ole kovin suosittuja kehittäjäyhteisössä, mikä laskee niiden kiinnostavuutta edelleen. Tässä raportissa keskitytään siis ensisijaisesti koontityökaluihin, joille on olemassa liitännäinen Jenkins CI:ssä.

3 Vaihtoehtoisten koontityökalujen evaluointi

Luvussa 2 tuotiin esille Antiin ja Maveniin kohdistuvaa kritiikkiä. Uusia, erityisesti Antin ja Mavenin puutteita ja ongelmia korjaavia koontityökaluja on kehitetty lukuisia. Tässä luvussa keskitytään uusiin koontityökaluihin, jotka eivät ole samankaltaisessa suosiossa kuin Ant ja Maven. Antia ja Mavenia käytetään myös ensisijaisina verrokkeina, mutta joitakin koontityökaluja on mielekästä vertailla esimerkiksi Makeen, joka on historiallisesti suosituin koontityökalu ja edelleen suuressa suosiossa. Koska Metropolian ohjelmistotuotantoprojekteissa kehitettävät projektit ovat usein Javalla tai PHP:llä kehitettäviä web-projekteja, syvennytään vertailussa erityisesti työkaluihin, jotka tarjoavat hyvän tuen Java-ekosysteemille ja PHP:lle. Tarkastelussa on myös muutamia yleiskäyttöisiä ja C- ja C++-kielillä tapahtuvaan kehitykseen erikoistuneita koontityökaluja.

Koontityökalujen vertailussa täytyy ottaa huomioon useita tekijöitä. Luonnollisesti suuri painoarvo asetetaan itse koontityökalun tarjoamille ominaisuuksille ja toiminnallisuudelle. Tämän lisäksi täytyy huomioida työkalun kypsyys, suosio ja kehityksen aktiivisuus. Kypsyyden arviointi objektiivisesti on äärimmäisen vaikeaa, mutta vihjeitä saadaan esimerkiksi uusimman vakaan julkaisun versionumerosta ja suosiosta erityisesti suurien, jäykempien organisaatioiden keskuudessa. Suurilla yrityksillä on usein huomattava kynnys ottaa käyttöön työkaluja, joilla ei ole takanaan pitkää ja menestyksekästä historiaa. Yritykset ovat usein myös kiinnostuneita maksamaan tuesta. Monilla korkean profiilin avoimen lähdekoodin projekteilla onkin takanaan yritys, joka tarjoaa työkaluun liittyviä palveluja. Tarvittaessa saatavilla olevat maksulliset palvelut lisäävät tuotteen uskottavuutta ja luottoa siihen. Koontityökalun kiinnostavuuteen vaikuttaa myös sen tuki kehitysympäristöissä. Käytännössä kaikki koontityökalut ovat komentorivisovelluksia, mutta useat integroidut kehitysympäristöt pystyvät kutsumaan niitä saumattomasti.

Taulukossa 1 on listattu Jenkinsin tukemia koontityökaluja. Lista ei ole täydellinen, mutta siinä on esitetty kaikki tämän työn kontekstissa kiinnostavat koontityökalut.

Taulukko 1. Jenkins CI:n tukemia koontityökaluja [Jenkins Plugins 2011].

Koontityökalu	Kuvauskieli	Uusin vakaa versio (julkaisupäivä)
Ant	XML	1.8.3 (29.2.2012)
CMake	makefile	2.8.7 (1.2.2012)
EasyAnt	XML	0.8 (10.8.2010)
Gant	Groovy Ant-skriptaus	1.9.7 (10.11.2011)
Gradle	Groovy DSL	1.0-rc-1 (11.4.2012)
Maven	XML	3.0.4 (20.1.2012)
MSBuild	XML	4.0 (3.3.2011)
NAnt	XML	0.91 (22.10.2011)
Phing	XML	2.4.12 (6.4.2012)
Rake	Ruby DSL	0.9.2.2 (22.10.2011)
Rational Application Developer Build Utility	XML	8.0 (20.8.2010)
Sbt	Scala DSL	0.11.1 (10.11.2011)
SCons	Python	2.1.0 (9.9.2011)
QMake	makefile	4.8.1 (28.3.2012)

Jenkinsille kehitetään jatkuvasti uusia liitännäisiä, joten tuettujen työkalujen lista muuttuu nopeaan tahtiin. Koontityökaluista on myös jonkinasteista ylitarjontaa, joten monien projektien kehitys on lakannut eikä niiden käyttämistä uusissa projekteissa voi suositella. Tällaisiin työkaluihin kuuluu muun muassa Kundo, jolle Jenkinsissä on tuki. Kundoa ei ole tästä syystä listattu taulukossa 1.

3.1 Rake – Ruby Make

Rake on Make-henkinen koontityökalu. Se käyttää sovellusaluekohtaisena kielenään Rubya ja se sisältyy Rubyn standardikirjastoon. Prosessikuvaustiedostoissa voi tarvittaessa hyödyntää kaikkia Rubyn ominaisuuksia ja rakenteita. Tämä tekee siitä huomattavasti joustavamman työkalun kuin esimerkiksi Ant. Antin toiminnallisuus rajoittuu tehtävien *(task)* suorittamiseen. Raken prosessikuvauksen yhteyteen voidaan liittää hyvin monimutkaista sovelluslogiikkaa, mikäli siihen on tarvetta. Kuvaustiedostot halutaan pitää yksinkertaisina, mutta koontityökalun käyttäjä voi Rakea käyttäessään olla varma siitä, ettei käytetyn kielen ilmaisuvoima lopu kesken missään tilanteessa.

Raken prosessikuvaukset ovat deklaratiivisia. Niissä kuvataan riippuvuussuhteita samalla tavalla kuin Antissa kuvataan päämäärien riippuvuuksia. Raken tarjoamat mahdollisuudet riippuvuussuhteiden määrittelyyn ovat kuitenkin huomattavasti joustavampia ja monipuolisempia kuin Antin yksinkertaiset mekanismit. Riippuvuussuhteessa saatetaan kuvata esimerkiksi käännetyn binääritiedoston riippuvuutta lähdekooditiedostoista. [Barnette 2009.] Koodiesimerkissä 4 on esitetty yksinkertainen prosessikuvaus, jossa hyödynnetään tiedostojen riippuvuuksia. Esimerkin prosessikuvaus tekee alkuperäisestä tiedostosta kopion, mikäli kopiota ei ole olemassa tai alkuperäistä tiedostoa on muokattu viimeisimmän kopioinnin jälkeen.

```
task :default => [:kopio]

file 'kopio' => 'alkuperainen' do |t|
    FileUtils.cp(t.prerequisites.first, t.name)
end
```

Koodiesimerkki 4. Rake-prosessikuvaus tiedostojen kopiointiin.

Prosessikuvausten keskeisimpiä komponentteja ovat tehtävät *(task)*. Tehtävät muistuttavat pitkälti Antin päämääriä, ja niiden välillä on usein riippuvuuksia. Rakessa erittäin keskeisiä tehtäviä ovat tiedostotehtävät *(file task)*, jotka muistuttavat enemmän Make-koontityökalun tehtäviä. Tiedostotehtävien tarkoituksena on tuottaa tiedosto. Tiedostotehtävät suoritetaan, mikäli määriteltyä tiedostoa ei ole olemassa tai sen syötetiedostoja eli riippuvaisuuksia on muokattu. Tehtävissä suoritettavaa koodia kutsutaan toiminnoksi *(action)*. [Fowler 2005.] Koodiesimerkin 4 tehtävä on tiedostotehtävä.

Raken selkeä puute on valmiin tuen puute yleisille kielille. Rake soveltuu hyvin Ruby-sovellusten koontiin. Se on erityisen kätevä esimerkiksi Ruby on Rails -sovelluskehystä käyttäviä sovelluksia kehitettäessä. Mikäli Rakea käyttää Java-projektin koonnin automatisointiin, kehittäjän täytyy itse kirjoittaa tehtävät esimerkiksi koodin kääntämiseen, yksikkötestien ajamiseen ja jar-pakkauksen luontiin. Tällaisten tehtävien kirjoittaminen vaatii jo huomattavasti Rubyn tuntemista. Raken tarjoamalla joustavuudella on siis hintansa; Raken prosessikuvauksien tekeminen vaatii runsaasti osaamista kehittäjältä, eikä käyttöönotto ole täten triviaalia. Java-projektien prosessikuvaukset voidaan kuitenkin pitää suhteellisen yksinkertaisina hyödyntämällä JRuby-projektia.

JRuby on Ruby-tulkki, joka on kirjoitettu Javalla. Se integroituu tiukasti Javaan ja mahdollistaa Ruby-koodin kutsumisen suoraan Java-ohjelmista. JRuby on Java-projektien koonnin kannalta mielenkiintoinen, sillä se mahdollistaa Antin hyödyntämisen Raken kautta. JRubyssä on Ant-kirjasto, jonka avulla Antin tehtäviä voi kutsua Raken prosessikuvauksissa. Antin tarjoamat tehtävät yksinkertaistavat erityisesti Java-koonnin prosessikuvausta huomattavasti, eikä niiden käyttö poista kehittäjältä Rubyn tuomia etuja. Ant-tehtäviä voi käyttää helposti intuitiivisella Rubyn syntaksilla. Koodiesimerkki 5 on yksinkertainen prosessikuvaus, jossa keskeisimmät Java-projektien tehtävät on implementoitu hyödyntämällä JRubyn Ant-kirjastoa. Prosessikuvauksessa on tehtävät koontituotteiden hakemiston poistamiseen, ohjelman kääntämiseen, paketointiin ja suorittamiseen.

```
require "ant"

PROJECT_NAME = "jruby-rake-demo"
JAR_NAME = "#{PROJECT_NAME}.jar"

SRC_DIR = "src/main/java"
BUILD_DIR = "target"
CLASSES_DIR = "#{BUILD_DIR}/classes"
SOURCE_COMPATIBILITY = "1.7"

task :default => [:run]

task :clean do
    ant.delete :dir => BUILD_DIR
end

task :compile do
    ant.mkdir :dir => CLASSES_DIR
    ant.javac :srcdir => SRC_DIR, :destdir => CLASSES_DIR,
      :source => SOURCE_COMPATIBILITY, :includeantruntime => "false"
end

task :jar => [:compile] do
```

```
    ant.jar :jarfile => "#{BUILD_DIR}/#{JAR_NAME}", :basedir => CLASSES_DIR do
       manifest do
           attribute :name => "Main-Class",
             :value => "fi.metropolia.esimerkki.Main"
       end
    end
end

task :run => [:jar] do
    ant.java :jar => "#{BUILD_DIR}/#{JAR_NAME}", :fork => "true"
end
```

Koodiesimerkki 5. Yksinkertaisen Java-projektin koonti Rakella käyttäen JRubya.

Rake on hyvin suosittu koontityökalu, koska se kuuluu Rubyn standardikirjastoon. Sen suosio on suurimmillaan Ruby-käyttäjien keskuudessa. Rakea voi käyttää yleispätevänä koontityökaluna, mutta monissa tapauksissa prosessikuvausten tekeminen on melko työlästä. Monet työkalut tarjoavat helpompia mekanismeja esimerkiksi Java-projektien koontiin.

3.2 Apache Buildr

Buildr perustuu Rakeen, jota käsiteltiin edellisessä luvussa. Kuten Rake se käyttää Rubyyn pohjautuvaa sovellusaluekohtaista kieltä koontiprosessin kuvaukseen. Buildr on kuitenkin filosofialtaan hyvin erilainen työkalu, ja se on tehty pääasiassa Java-sovellusten koontiin. Kuten Maven, Buildr on koonnin sovelluskehys, jolla projekteja voidaan hallita kokonaisvaltaisesti toisin kuin Rakella. Buildr tarjoaa siis suosituimpien riippuvuuksienhallintatyökalujen kanssa yhteensopivan riippuvuuksienhallinnan ja modulaarisen arkkitehtuurin, joka mahdollistaa liitännäisten tekemisen mihin tahansa tarpeeseen. Liitännäiset mahdollistavat yksinkertaisten ja ymmärrettävien prosessikuvausten tekemisen ilman, että ilmaisuvoimaa menetetään. Buildrin prosessikuvaukset ovatkin usein lyhytsanaisempia kuin esimerkiksi Antin vastaavan toiminnallisuuden tarjoavat XML-pohjaiset kuvaukset. [Buildr.]

Buildr on ensisijaisesti Java-ekosysteemin kielten koontiin tarkoitettu. Se tukee hyvin muun muassa Javaa, Scalaa ja Groovya. Buildrin riippuvuuksienhallinta on yhteensopiva Maven-tietovarastojen kanssa, ja liitännäisen avulla myös Ivy-tietovarastot ovat tuettuina. Buildrin prosessikuvausten oletusarvot on tehty Java-projekteja varten. Tämän ansiosta Java-projektien prosessikuvaustiedostot ovat Buildrillä usein erittäin

kompakteja. Koodiesimerkin 6 prosessikuvaus on erittäin yksinkertainen Java-projektin prosessikuvaustiedosto.

```
define 'esimerkki' do
    project.version = '1.0.0'
    package(:jar).with :manifest=>{
        "Main-Class" => "fi.metropolia.esimerkki.Main"
    }
end
```

Koodiesimerkki 6. Buildr-prosessikuvaus Java-sovelluksen koontiin.

Yksinkertaisuudestaan huolimatta kyseinen prosessikuvaus tarjoaa lukuisia tavallisesti Java-kehityksessä tarvittavia tehtäviä. Näihin kuuluvat luonnollisesti koodin kääntäminen ja paketointi, mutta myös yksikkötestien ajaminen, suosituimpien integroitujen kehitysympäristöjen projektitiedostojen generointi ja useita muita erittäin hyödyllisiä tehtäviä. Koodiesimerkissä esitetty prosessikuvaus toimii, mikäli projekti käyttää standardiksi muodostunutta kuvassa 2 esitettyä hakemistorakennetta. Muussa tapauksessa prosessikuvaukseen täytyy lisätä määrittelyjä, jotka kertovat Buildrille projektin hakemistorakenteen.

Buildr keskittyy enemmän helppokäyttöisyyteen kuin Rake. Buildr-prosessikuvausten tekeminen on täysin mahdollista ilman Rubyn osaamista. Tämä onkin tärkeää, koska Ruby ei ole Java-kehittäjille välttämättä tuttu ohjelmointikieli. Antin käyttäjät voivat helposti siirtyä Buildrin käyttäjiksi, sillä Buildrilla voi kutsua Antin tehtäviä. Buildr käyttää Antin integroimiseen Antwrap-moduulia [Antwrap 2008]. Rubyn osaaminen tulee kuitenkin tarpeeseen, kun prosessikuvauksissa tarvitaan sellaista toiminnallisuutta, mitä Buildrin ydin ja olemassa olevat liitännäiset eivät tarjoa. Rubyllä omien tehtävien tekeminen on helppoa ja tapahtuu täysin samalla tavalla kuin Rakessa, koska Buildr perustuu Rakeen. Raken käyttäjät voivatkin helposti siirtyä Buildrin käyttäjiksi.

Kuten edellä todettiin, Buildrin prosessikuvauksiin voi liittää mitä tahansa sovelluslogiikkaa. Yleensä prosessikuvauksiin lisätään tehtäviä ja mahdollisesti funktioita. Funktiot ovat usein uudelleenkäytettäviä, eikä niitä välttämättä haluta pitää näkyvillä prosessikuvauksessa. Yksinkertaisimmillaan uudelleenkäytettävät tehtävät tai funktiot voi kirjoittaa omiin tiedostoihinsa, jotka merkataan rake-tiedostopäätteellä. Buildr hakee näitä tiedostopäätteitä prosessikuvaustiedoston hakemistossa sijaitsevasta tasks-alihakemistosta. Tasks-hakemisto voidaan viedä versionhallintaan tai se voidaan jakaa

jollain muulla tavalla. Varsinaiset liitännäiset ovat Buildrissä Ruby-moduuleja, jotka hyödyntävät Buildrin ohjelmointirajapintaa. Buildrin jakeluun kuuluvat keskeiset tehtävät on toteutettu ohjelmoimalla tätä rajapintaa vasten. Tehtävien, funktioiden ja liitännäisten jakeluun ei ole Buildrissä standardoitua tapaa. Versionhallinnan käyttäminen soveltuu lähinnä yrityksen sisäiseen käyttöön tarkoitettujen liitännäisten jakamiseen. Yksi hyvä tapa laajempaa jakelua ajatellen on pakata liitännäinen RubyGems-paketinhallintajärjestelmän ymmärtämäksi paketiksi eli *gemiksi*. RubyGems on paketinhallintajärjestelmä Ruby-kielelle. Sen avulla Ruby-ohjelmien ja -kirjastojen jakelu ja asentaminen onnistuu helposti.

Buildr tarjoaa samankaltaisen elämänkaarimallin kuin Maven. Kuten koodiesimerkki 6 osoittaa, tyypillisimpien koontitehtävien käyttöönotto onnistuu hyvin minimaalisella prosessikuvaustiedostolla. Koodiesimerkin prosessikuvaus luo automaattisesti tehtävät koodin kääntämiseen ja yksikkötestien suorittamiseen. Lisäksi Buildr luo jokaiselle projektille build-tehtävän, joka sekä suorittaa yksikkötestit että kääntää koodin [Buildr::Project.] Build on oletustehtävä, joka suoritetaan, kun Buildriä kutsutaan ilman argumentteja.

Buildr on melko vakuuttava koontityökalu, joka onnistuu yhdistämään Antin ja Mavenin hyvät puolet ja poistamaan useita epäkohtia. Sen prosessikuvaukset pysyvät usein hyvin luettavina ja loogisina. Buildrin laajennusmahdollisuudet ovat myös hyvät. Se ei kuitenkaan ole varteenotettava vaihtoehto Metropolian ohjelmistotuotantoprojekteissa käytettäväksi koontityökaluksi; Jenkins CI -koontipalvelimelle ei ole olemassa liitännäistä, jolla Buildr voitaisiin liittää osaksi koontisilmukkaa. Tämän lisäksi Ruby on useimmille opiskelijoille tuntemattomampi kieli kuin Java-perheeseen kuuluvat kielet. Ruby on helposti omaksuttava ja syntaksiltaan tehokas kieli, mutta mikäli itse projektissa ei käytetä Rubya, on sen opettelua vaikea perustella pelkästään koontityökalua varten.

3.3 SCons

SCons on yleiskäyttöinen koontityökalu, joka käyttää prosessikuvauskielenä Pythonia. SCons noudattaa siis nykyistä trendiä, jossa koontityökalun prosessikuvauskieli perustuu yleiskäyttöiseen ohjelmointikieleen. Pythonin käyttö tarkoittaa sitä, että prosessikuvauksien mahdollisuudet rajoittuvat lähinnä ohjelmoijan taitoihin. Joustavuuden ansios-

ta SConsia voi käyttää hyvin erilaisten projektien koontiin ohjelmointikielestä ja sovellusalustasta riippumatta. Lisäksi sen käyttämä syntaksi on jo jossain määrin tuttua monille ohjelmoijille, koska Python on erittäin suosittu kieli.

SCons on saavuttanut suosiota lähinnä C- ja C++-ohjelmoijien keskuudessa. Monet avoimen lähdekoodin projektit käyttävät sitä, ja valtaosa niistä on C:llä tai C++:lla toteutettuja [Scons Projects 2011]. SConsissa on sisäänrakennettu tuki myös monille muille kielille, kuten D:lle, Javalle ja Fortranille. SConsin parhaiten tuetut kielet C:n ja C++:n lisäksi ovat Fortran ja LaTex. Kehitystyö muiden kielien tukemiseksi on ollut vähäistä ja monet muut koontityökalut tukevat esimerkiksi Java-ekosysteemin kieliä huomattavasti paremmin. Käyttäjäyhteisössä onkin esitetty näkemyksiä, joiden mukaan SConsin kannattaisi keskittyä vankistamaan asemaansa koontityökaluna C- ja C++-projekteille ja jättää Java-tuki muiden koontityökalujen tehtäväksi. [Java Strategy 2008; Java Support 2010.]

C- tai C++-projekteja ei ole Metropolian ohjelmistotuotantoprojekteissa juurikaan kehitetty jatkuvan integroinnin ympäristön perustamisen jälkeen. Ympäristöä pyritään edelleen ensisijaisesti kehittämään Java-ekosysteemin ja suosittujen web-teknologioilla toteutettujen projektien koontiin. SConsin vahvuudet ovat muualla, joten se ei sovellu hyvin usempiin Metropolian ohjelmistoprojekteihin. SCons on kuitenkin Jenkinsissä hyvin tuettu, joten se on harkitsemisen arvoinen koontityökalu C- ja C++-projekteille.

3.4 Gant

Gant ei ole täydellinen koonnin sovelluskehys, vaan se mahdollistaa Antin käyttämisen Groovy-skriptikielellä. Groovy on dynaaminen ohjelmointikieli, joka muistuttaa Pythonia, Rubya ja Perliä. Ant-tehtävien lisäksi Gantin prosessikuvauksissa voi vapaasti käyttää Groovya koontilogiikan ohjelmointiin. Käytännössä Gant on siis pitkälti vaihtoehtoinen tapa käyttää Antia. Sillä voi suorittaa koonnin vaiheita, mutta siinä ei ole esimerkiksi riippuvuuksienhallintaa ja elämänkaarimallia. Yksinkertaisen Java-projektin koonnin prosessikuvaus on esitetty koodiesimerkissä 7. Se muistuttaa hyvin paljon Raken vastaavan toiminnallisuuden tarjoavaa prosessikuvausta. Erona onkin lähinnä Groovyn ja Rubyn syntaktisest eroavaisuudet.

```
projectName = "gant-demo"
jarName = "${projectName}.jar"

sourceDirectory = "src/main/java"
buildDirectory = "target"
classesDirectory = "${buildDirectory}/classes"

includeTargets << gant.targets.Clean
cleanPattern << "**/*~"
cleanDirectory << buildDirectory

target(compile: "Käännetään lähdekoodi") {
    mkdir(dir: classesDirectory)
    javac(srcdir: sourceDirectory, destdir: classesDirectory,
      includeantruntime: false)
}

target(jar: "Tehdään jar-paketti") {
    depends(compile)
    jar(jarfile: "${buildDirectory}/${jarName}", basedir: classesDirectory) {
        manifest() {
            attribute(name: "Main-Class", value: "fi.metropolia.esimerkki.Main")
        }
    }
}

target(run: "Ajetaan sovellus") {
    depends(jar)
    java(jar: "${buildDirectory}/${jarName}", fork: true)
}
```

Koodiesimerkki 7. Gant-prosessikuvaus Java-sovelluksen koontiin.

Seuraavassa luvussa käsiteltävän Gradlen oli alun perin tarkoitus hyödyntää Gantia, mutta Gradlen kehittäjät päätyivät aloittamaan koontityökalunsa rakentamisen puhtaalta pöydältä [Winder 2008]. Huomattavaa on myös, että nykyisin Gant-projekti käyttää koontityökalunaan Gradlea. Gradlellä pystytään tekemää kaikki samat asiat kuin Gantilla menettämättä eleganttia syntaksia ja muita Gantin hyviä puolia. Gant on siis jäänyt lähinnä kuriositeetiksi.

3.5 Gradleware Gradle

Gradle on erityisesti Java-ekosysteemin kielille tarkoitettu moderni koontityökalu. Gradlen prosessikuvaukset tehdään Groovy-pohjaisella sovellusaluekohtaisella kielellä. Gradlen käyttämä kieli muistuttaa jonkin verran Gantin käyttämää kieltä, koska Gradle on saanut vahvoja vaikutteita Gantista. Toisin kuin Gantissa, Gradlessä on integroituna riippuvuuksienhallinta ja sitä voidaankin kutsua täydeksi projektinhallintatyökaluksi. Gradle on toiminnallisuudeltaan hyvin lähellä Mavenia ja siihen vertailu on mielekästä.

Gradle on selkeästi tehty Mavenin kilpailijaksi ja korvaajaksi; sen suunnittelussa on panostettu erityisesti Mavenin heikkouksien korjaamiseen. Mavenista poiketen Gradleä kutsutaan myös koonnin integrointityökaluksi. Gradle mahdollistaa Ant-tehtävien ja jopa täydellisten Ant-prosessikuvausten käyttämisen koonnissa. Lisäksi Gradle tukee täysin sekä Ivy- että Maven-tietovarastoja.

Gradlen prosessikuvauksissa voi hyödyntää kaikkia Groovyn ominaisuuksia. Suurin osa Javan syntaksista on suoraan käytettävissä Groovyssa. Javan syntaksi on kuitenkin monin paikoin huomattavasti monisanaisempaa, joten Groovyn syntaksin opettelu on suositeltavaa, mikäli imperatiiviseen koontilogiikan ohjelmointiin on tarvetta. Koodiesimerkki 8 havainnollistaa Groovyn ilmaisuvoimaa.

```
defaultTasks 'evenNumbers'

task evenNumbers << {
    println ((1..10).findAll { it % 2 == 0 })
}
```

Koodiesimerkki 8. Gradle-prosessikuvaus, joka tulostaa parilliset luvut yhden ja kymmenen väliltä.

Esimerkin toiminnallisuus on sama kuin koodiesimerkin 2 Ant-prosessikuvauksessa. Lyhyydestään huolimatta se on helpommin ymmärrettävissä kuin Antilla tehty prosessikuvaus. Esimerkki voitaisiin esittää myös Javan monisanaisemmalla syntaksilla. Se olisi silti ohjelmoijille huomattavasti helpompi sisäistää kuin Antin XML-muotoinen prosessikuvaus.

Vaikka Gradlellä on tarvittaessa mahdollista ohjelmoida koontilogiikkaa imperatiivisesti, tavallisesti Gradlen prosessikuvaukset ovat deklaratiivisia. Gradlen prosessikuvaukset perustuvat samantapaisiin käytäntöihin kuin Mavenin projektioliomalli. Prosessikuvauksiin ei siis eksplisiittisesti kuvata kaikkia koonnin tapahtumia. Tätä havainnollistaa hyvin koodiesimerkki 9.

```
apply plugin: 'java'

defaultTasks 'clean', 'build'

sourceCompatibility = 1.7

jar {
    manifest { attributes("Main-Class": "fi.metropolia.esimerkki.Main") }
}
```

Koodiesimerkki 9. Gradle-prosessikuvaus Java-sovelluksen koontiin.

Esimerkin prosessikuvaus kääntää projektin lähdekoodin, suorittaa yksikkötestit ja tekee projektista ajettavan jar-pakkauksen. Koonti onnistuu, mikäli projektin hakemistorakenne noudattaa Mavenin standardoimaa hakemistorakennetta, joka on esitetty kuvassa 2. Mavenin ja Buildrin vastaavat prosessikuvaukset ovat hyvin samankaltaisia, joskin erilaisella syntaksilla ilmaistuna. Näiden kolmen koontityökalun väliset erot tulevatkin paremmin esiin monimutkaisemmassa käytössä.

Gradlen suunnittelussa on erityisesti otettu huomioon useammista projekteista ja moduuleista koostuvat koontityöt. Monille suurille sovelluksille tämä on hyvin tärkeää. Suuret projektit hajautetaan usein moduuleihin, joilla on oma kehityselämänkaarensa. Moduuleista koostuva projekti täytyy kuitenkin integroida yhdeksi testattavaksi ja julkaistavaksi kokonaisuudeksi. Gradle on erityisen hyvä koontityökalu tällaisia projekteja varten. Hibernate-projekti on yksi Mavenista Gradleen siirtyneistä korkean profiilin projekteista. Projektin pääkehittäjä Steve Ebersole [2010] on maininnut yhdeksi siirtymisen tärkeimmiksi syiksi Gradlen paremman tuen useista moduuleista koostuville projekteille. Gradle tukee hyvin myös monitasoisia projekteja, joissa aliprojekteilla voi olla omia aliprojektejaan. Koodiesimerkissä 10 on esitetty erittäin yksinkertainen kahdesta projektista koostuva Java-koonti.

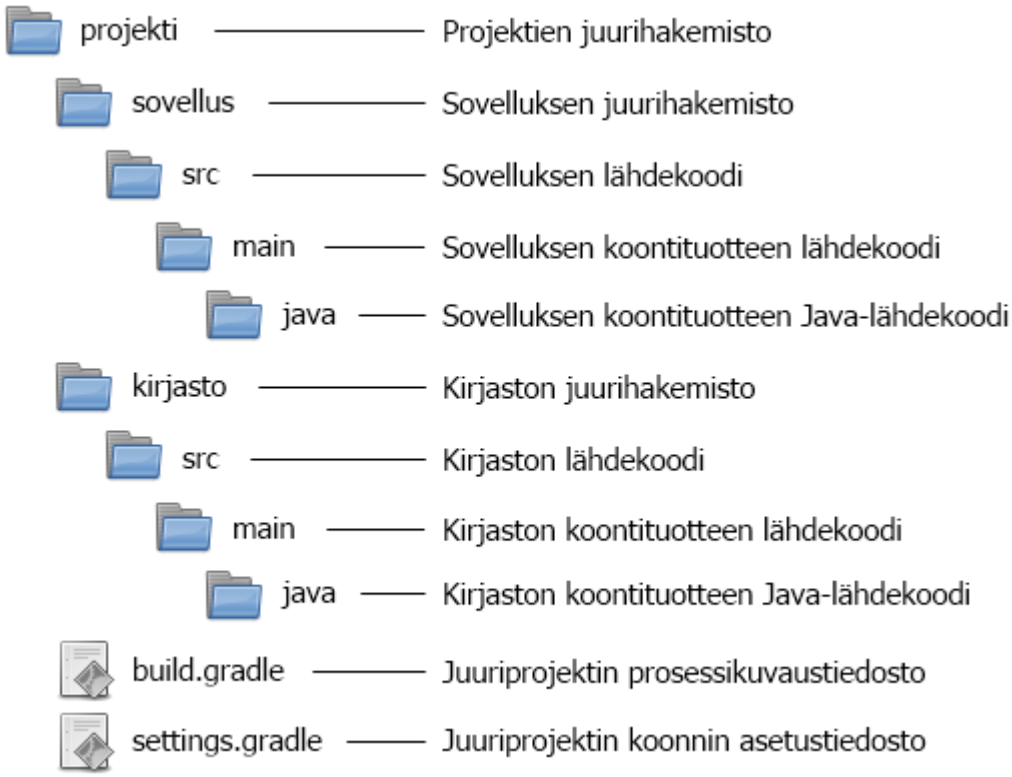
```
// settings.gradle
include 'sovellus', 'kirjasto'

// build.gradle
defaultTasks 'clean', 'build'

subprojects {
    apply plugin: 'java'
    sourceCompatibility = 1.7
}

project(':sovellus') {
    dependencies {
        compile project(':kirjasto')
```

```
    }

    jar {
        from { configurations.compile.collect { it.isDirectory() ? it : zipTree(it) }
}
        manifest { attributes("Main-Class": "fi.metropolia.esimerkki.Main") }
    }
}
```

Koodiesimerkki 10. Kahdesta projektista koostuva Gradle-prosessikuvaus.

Kyseessä on Java-ympäristölle tyypillinen projektihierarkia, jossa toinen projekteista on yleiskäyttöinen kirjasto ja toinen sovellus, joka riippuu kyseisestä kirjastosta. Esimerkissä koko koonti on kuvattu juuriprojektin prosessikuvaustiedostossa. Gradle mahdollistaa myös prosessikuvauksen hajauttamisen siten, että projektikohtaiset kuvaukset tehdään projektien omissa prosessikuvaustiedostoissa. Gradlen juuriprojekti vaatii asetustiedoston, joka kertoo Gradlelle, missä alihakemistoissa projektit sijaitsevat. Koodiesimerkissä 10 tämä tiedosto on settings.gradle.

Kuva 3 havainnollistaa hakemistorakennetta, jota käytetään koodiesimerkissä 10.



Kuva 3. Kahdesta projektista koostuvan Gradle-koonnin hakemistorakenne.

Projektista on jätetty yksinkertaistamisen vuoksi pois muun muassa testit, jotka sijoitettaisiin omiin hakemistoihinsa. Tärkeää on huomata esimerkissä esitetyn settings.gradle-tiedoston sisältö. Asetustiedostossa esitellään hakemistot, joista aliprojektit löytyvät.

Kuten edellä todettiin, Gradleä voidaan kutsua koonnin integrointityökaluksi. Antin käyttäminen onnistuu Gradlen kautta samankaltaisella syntaksilla kuin Gantin prosessikuvauksissa. Gradle vie kuitenkin Antin integroimisen pidemmälle kuin Gant. Gradlen prosessikuvauksissa voi käyttää täyttä Ant-prosessikuvausta sellaisenaan koonnin osana. Koodiesimerkissä 1 esitettiin yksinkertainen Ant-prosessikuvaustiedosto, jolla voi koota Java-sovelluksen. Koodiesimerkki 11 hyödyntää kyseistä Ant-prosessikuvausta.

```
ant.importBuild 'build.xml'
```

Koodiesimerkki 11. Ant-prosessikuvauksen tuominen Gradleen.

Koko prosessikuvaus saadaan tuotua Gradleen helposti yhdellä rivillä. Tämän jälkeen kaikki Antin prosessikuvauksen päämäärät *(target)* ovat käytettävissä aivan kuin ne olisi Gradlen tehtäviä *(task)*. Gradleen on suunnitteilla tuki myös Mavenin projektioliomallien hyödyntämiselle. Mavenista Gradleen siirtyminen on kuitenkin suhteellisen helppoa jo nykyisin, koska Gradle tukee täysin Mavenin rakennetta käyttäviä tietovarastoja ja hyödyntää samoja käytäntöjä kuin Maven.

Koontityökalujen integroituminen Gradleen voi kuulostaa turhalta ominaisuudelta, koska Antin ja Mavenin koontilogiikka voidaan yleensä toteuttaa helpommin ja yksinkertaisemmin Gradlella. Uudet ja vasta koontityökalua valitsemassa olevat projektit eivät ominaisuudesta yleensä hyödykään. Lähes kaikki vakavasti otettavat projektit, jotka ovat olleet kehityksessä pidemmän aikaa, käyttävät jotain koontityökalua. Java-projektien tapauksessa koontityökalu on useimmiten Ant tai Maven. Antia käyttävien projektien on erittäin helppo siirtyä vaiheittain käyttämään Gradlea, koska koko koontiprosessia ei tarvitse suoraan muuntaa Gradlen sovellusaluekohtaista kieltä käyttäväksi. Suurien projektien koontilogiikka saattaa olla hyvinkin monimutkaista, jolloin uuden koontityökalun käyttöönotto saattaa viedä paljon kehitysresursseja. Tällöin koontityökalua ei helposti ryhdytä vaihtamaan, vaikka sillä saavutettaisiin etuja vanhaan työkaluun verrattuna.

Gradlen ympärillä ei ole vielä yhtä suurta ekosysteemiä kuin Antilla ja erityisesti Mavenilla. Liitännäistarjonta on jonkin verran vaatimattomampaa ja useat aktiivisesti kehitetyt liitännäiset ovat yhden kehittäjän työpanoksen tuloksia. Liitännäisiä on kuitenkin saatavilla moniin tarkoituksiin, ja esimerkiksi Android-kehitys onnistuu Gradlellä. Tätä tarkastellaan tarkemmin luvussa 5.1. Integroidut kehitysympäristöt eivät tue Gradleä aivan yhtä hyvin kuin Antia tai Mavenia. Tilanne on kuitenkin jatkuvasti paranemassa. Eclipse, IntelliJ IDEA ja NetBeans sisältävät jo jonkinasteisen tuen Gradlelle. Uusimassa julkaistussa IntelliJ IDEAn versiossa on Gradle-liitännäinen valmiiksi integroituna, joten se tarjoaa kenties helpoimman tien Gradlen käyttöön integroidun kehitysympäristön sisältä.

Gradle ei tyydy pelkästään parantamaan Antin ja Mavenin olemassa olevia ominaisuuksia. Gradlen ominaisuuspalettiin kuuluu myös monia hyvin hyödyllisiä ominaisuuksia, jotka löytyvät vain harvoista koontityökaluista. Esimerkkejä tällaisista ominaisuuksista on muun muassa Gradle Daemon ja Gradle Wrapper. Gradle Daemon on taustaprosessi, joka nopeuttaa Gradlen koonnin suoritusaikaa uudelleenkäytettävän taustaprosessin avulla. Gradle Daemonia voi jo käyttää, mutta se on merkattu eksperimentaaliseksi ominaisuudeksi, eli täysiä takeita sen ongelmattomasta toimivuudesta ei anneta. Gradle Daemonin kehityksessä on kuitenkin lupauduttu keskittymään moniin ominaisuuksiin, jotka saattavat tehdä siitä suositellun tavan käyttää Gradleä. [Dockter & Murdoch 2012b.]

Gradle Wrapper mahdollistaa itse koontityökalun hallitsemisen ulkoisen riippuvuuden tavoin. Wrapper osaa tarvittaessa ladata prosessikuvauksen määrittelemän Gradlen version, mikäli sitä ei ole suoritusympäristössä valmiiksi saatavilla. Tämä on erityisen kätevä ominaisuus koontipalvelimille, koska niille ei tarvitse asentaa tiettyä Gradlen versiota, vaan oikea versio haetaan automaattisesti. Lisäksi Wrapper helpottaa kehitystä, koska yksittäisten kehittäjien ei tarvitse asentaa Gradleä. Kaikilla projektin kehittäjillä on myös automaattisesti käytössään sama Gradlen versio, joka määritellään prosessikuvaustiedostossa. Yhteensopivuusongelmia ei siis pääse syntymään, ja työkalun päivitys haluttuun versioon tapahtuu heti, kun kehittäjä hakee versionhallinnasta päivitetyn prosessikuvauksen ja suorittaa Gradle Wrapperin ensimmäisen kerran päivityksen jälkeen. [Dockter & Murdoch 2012c.] Koodiesimerkissä 12 on esitetty prosessikuvaus Gradle Wrapperin käyttöönottoon.

```
task wrapper(type: Wrapper) {
    gradleVersion = '1.0-milestone-8'
}
```

Koodiesimerkki 12. Tehtävä Gradle Wrapperin asentamiseen.

Gradle on erittäin hyvä yleiskäyttöinen koontityökalu. Gradlen kehittäjät ovat tunnistaneet Antin ja Mavenin hyvät ja huonot puolet ja kehittäneet koontityökalun, joka pyrkii yhdistämään näiden työkalujen hyvät puolet toimivaksi kokonaisuudeksi. Groovyn kaltaisen skriptikielen käyttäminen koontilogiikan ohjelmoimiseen on huomattavasti luontevampaa kuin XML-pohjaisten sovellusaluekohtaisten kielien käyttäminen. Gradle tarjoaa joustavuuden lisäksi tuen Mavenin standardoimille käytännöille, joten prosessikuvaukset saadaan yleensä pidettyä tiiviinä ja informatiivisina. Gradle on kuitenkin huomattavasti joustavampi työkalu kuin Maven. Ennen kaikkea Gradle antaa kehittäjälle täyden vapauden koontilogiikan ohjelmointiin. Maven pakottaa käyttäjän ajattelemaan koontilogiikkaa Mavenin määrittelemien hyvin tarkkojen rajoitusten pohjalta. Gradlen prosessikuvauksen ohjelmointi on vain yksi ratkaistava ohjelmointiongelma muun kehityksen lomassa. Tämän vuoksi Gradle on erittäin hyvä yleispätevä koontityökalu.

3.6 Phing

Phing on Antiin pohjautuva koontityökalu, joka on tarkoitettu PHP-sovellusten koontiin. Se ei ole siis yleiskäyttöinen koontityökalu, kuten monet muut edellä esitetyistä työkaluista. Phing ei hyödynnä Antin lähdekoodia, vaan se on ohjelmoitu kokonaan PHP:lla. Phingin ja Antin prosessikuvaukset ovat kuitenkin syntaksiltaan täysin samanlaisia. Phing tarjoaa valmiit tehtävät yleisimpien PHP-projekteissa käytettyjen laaduntarkastustyökalujen käyttöön. Tällaisia työkaluja ovat muun muassa PHPUnit, DocBlox ja PHP CodeSniffer. Työkaluilla voi esimerkiksi suorittaa yksikkötestejä, generoida dokumentaatiota ja analysoida koodia staattisesti. PHP-laaduntarkastustyökaluja käsitellään tarkemmin luvussa 6.1. Phingillä voi myös helposti hyödyntää PEAR-julkaisujärjestelmää. PHP-sovellusten julkaisussa on usein tarpeen alustaa tietokanta ja muokata tiedostojen ja hakemistojen oikeuksia. Phingissä on valmiit tehtävät myös näihin tarpeisiin.

Koodiesimerkki 13 on minimaalisen PHP-projektin koontiin tarkoitettu prosessikuvaus.

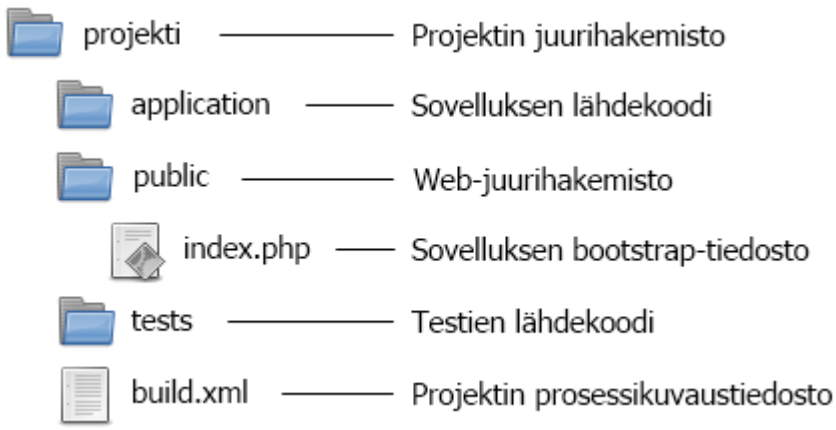
```
<?xml version="1.0" encoding="UTF-8"?>

<project name="phing-demo" default="phpqa">

    <fileset id="sources" dir="application">
        <include name="**/*.php"/>
    </fileset>

    <property name="testsDir" value="tests"/>
    <property name="buildDir" value="build"/>

    <target name="phpqa" depends="clean,test,phpmd"
      description="Suorittaa laaduntarkastustyökalut"/>

    <target name="clean">
         <delete dir="${buildDir}"/>
    </target>

    <target name="init">
        <mkdir dir="${buildDir}"/>
    </target>

    <target name="coverage-db" depends="init"
      description="Luo tietokannan testikattavuuksia varten">
        <coverage-setup database="${buildDir}/coverage.db">
            <fileset refid="sources"/>
        </coverage-setup>
    </target>

    <target name="test" depends="coverage-db"
      description="Suorittaa yksikkötestit">
        <phpunit bootstrap="${testsDir}/bootstrap.php" haltonfailure="true"
          haltonerror="true" codecoverage="true">
            <formatter todir="${buildDir}" type="xml"/>
            <formatter todir="${buildDir}" type="clover"/>
            <batchtest>
                <fileset dir="${testsDir}">
                    <include name="**/*Test*.php"/>
                    <exclude name="**/Abstract*.php"/>
                </fileset>
            </batchtest>
        </phpunit>
    </target>

    <target name="phpmd" depends="init"
      description="Suorittaa Project Mess Detectorin">
        <phpmd>
            <fileset refid="sources"/>
            <formatter type="xml" outfile="${buildDir}/pmd.xml"/>
         </phpmd>
     </target>

</project>
```

Koodiesimerkki 13. Phing-prosessikuvaus PHP-projektin koontiin.

Esimerkkisovelluksen yhteydessä koonnilla ei tarkoiteta aivan samaa asiaa kuin esimerkiksi tyypillisen Java-sovelluksen tapauksessa. PHP-koodia ei tavallisesti käännetä eikä pakata, vaan koodi tulkataan web-palvelimella. Esimerkin prosessikuvaus ajaa yksikkö-

testit ja generoi niiden pohjalta testikattavuusraportin. Lisäksi se suorittaa PHP Mess Detector -työkalun. Työkalut tuottavat XML-muotoisia raportteja, joita voi hyödyntää esimerkiksi Jenkins CI -koontipalvelimella.

PHP-projekteille ei ole standardoitunutta hakemistorakennetta, mutta koodiesimerkin 13 prosessikuvaus on tehty laajalti käytetylle modernille hakemistorakenteelle, jollaista myös monet suosituimmat PHP-sovelluskehykset käyttävät. Hakemistorakenne on esitetty kuvassa 4.



Kuva 4. Minimaalisen PHP-projektin hakemistorakenne.

Rakenne on yksinkertaistettu äärimmilleen, eikä siinä ole monia sovelluskehyksien käyttämiä hakemistoja. Kuvatun kaltaisessa rakenteessa sovelluksen lähdekoodia ei laiteta web-palvelimen kautta saataville, vaan palvelinsovelluksen tai siihen konfiguroidun virtuaalipalvelimen juurihakemisto asetetaan osoittamaan public-hakemistoon. Tässä hakemistossa on ainoastaan sovelluksen käynnistävä bootstrap-tiedosto, joka on tyypillisesti nimeltään index.php.

Phingin tarjoamat valmiit PHP-projekteja varten tarkoitetut tehtävät ovat suurimmilta osin toteutettavissa Antilla. Phing ei siis mullista PHP-sovellusten koontia, vaan se tarjoaa hieman helppokäyttöisemmän syntaksin PHP-projekteissa yleisesti käytettyjen laaduntarkastustyökalujen liittämiseksi koontiprosessiin. Syntaksi on myös Antin käyttäjille tuttu, joten Antista Phingiin siirtyminen on hyvin vaivatonta. Toisaalta monet Phingiin sisäänrakennetut tehtävät ovat saatavilla Antiin liitännäisinä. Antin ympärillä on erittäin laaja ekosysteemi, koska se on ollut pitkään erittäin suosittu koontityökalu mo-

nissa eri ympäristöissä. Sitä voidaan melko helposti hyödyntää myös PHP-projektien koontiin.

Kuten lähes kaikkia koontityökaluja, myös Phingin toiminnallisuutta voi laajentaa liitännäisillä. Phingin liitännäiset kirjoitetaan PHP-kielellä. Mahdollisuus ohjelmoida liitännäisiä tutulla kielellä onkin kenties Phingin suurin valtti. Antin liitännäiset täytyy toteuttaa Javalla, joka ei ole välttämättä PHP-kehittäjille tuttu kieli. Phingiä käyttämällä Javan opiskeluun ei ole tarvetta. Mikäli PHP-projektin koonti vaatii siis toiminnallisuutta, jota koontityökalut eivät valmiina tarjoa, ja PHP-kieli on kehittäjäryhmän vahvinta osaamisaluetta, Phing voi olla hyvä valinta koontityökaluksi.

3.7 Sbt

Sbt (Simple Build Tool) on tarkoitettu erityisesti Scala- ja Java-projektien koontiin. Sbt:n prosessikuvaukset tehdään Scalaan pohjautuvalla sovellusaluekohtaisella kielellä. Sbt:n prosessikuvauksissa voi käyttää Scalaa sellaisenaan koontilogiikan ohjelmointiin. Tämä piirre on yhteinen kaikille moderneille koontityökaluille, joiden sovellusaluekohtainen kieli perustuu Turing-täydelliseen ohjelmointikieleen. Scala on moderni yleiskäyttöinen ohjelmointikieli, jolla voi ohjelmoida oliopohjaisesti ja funktionaalisesti. Kehittäjällä on siis koontilogiikan ohjelmoinnissa vapaat kädet, eikä kielen ilmaisuvoima lopu kesken. Sbt on sukua Mavenille, Buildrille ja Gradlelle. Se on erityisesti Java-ekosysteemin ohjelmointikieliä varten tehty ja se hyödyntää Mavenin standardoimia käytäntöjä. Sbt:ssä on myös sisäänrakennettu riippuvuuksienhallinta, joka on toteutettu Apache Ivylla. Suurimman suosion ja käyttäjäkunnan Sbt on saanut Scala-kehittäjien keskuudessa. Sbt on myös ohjelmoitu Scalalla.

Sbt:n prosessikuvaukset pysyvät tiiviinä Scalaan pohjautuvan sovellusaluekohtaisen kielen ansiosta. Syntaksi on hyvin eleganttia, mutta ei aivan helposti ymmärrettävää kokemattomalle ja Scalaa tuntemattomalle käyttäjälle. Koodiesimerkissä 14 on esitetty hyvin minimaalinen prosessikuvaus Java-sovelluksen koontiin.

```
name := "hello"

version := "1.0"

organization := "fi.metropolia"
```

```
javacOptions ++= Seq("-source", "1.7", "-target", "1.7")

crossPaths := false
```

Koodiesimerkki 14. Prosessikuvaus yksinkertaisen Java-sovelluksen koontiin

Tällä prosessikuvauksella sovellus voidaan kääntää, testata, paketoida ja suorittaa. Lisäksi käytettävissä on muita tehtäviä *(task)* muun muassa riippuvuuksienhallintaa varten. Huomionarvoista on rivinvaihtojen käyttäminen prosessikuvaustiedostossa. Sbt:n käyttämässä kielessä asetukset erotetaan toisistaan tyhjällä rivillä niiden välissä. Prosessikuvauksesta nähdään myös Sbt:n juuret Scala-koontityökaluna, sillä ilman crossPaths-asetuksen käytöstä poistamista hakemistopolkuihin ja koontituotteiden nimiin liitetään päätteeksi käytetyn Scala-version versionumero.

Sbt:n suorittamiseen on useita tapoja. Sen voi ajaa samaan tapaan kuin lähes kaikki koontityökalut, eli kutsumalla komentoriviltä ja antamalla koontitehtävät parametreina. Useimmista koontityökaluista poiketen Sbt:ssä on interaktiivinen tila, jossa komentoja voi kutsua helposti. Interaktiivisessa tilassa on käytössä monista komentotulkeista tutut ominaisuudet, kuten komentohistoria ja komentojen täydentäminen sarkaimella. Kenties mielenkiintoisin ja koontityökalujen piirissä lähes uniikki vaihtoehto koonnin suorittamiseen on Sbt:n jatkuva koonti. Tässä tilassa Sbt suorittaa koonnin automaattisesti aina kun muokatut lähdekooditiedostot tallennetaan. Tämä nopeuttaa kehitystä, koska kehittäjän ei tarvitse erikseen kutsua koontityökalua ja palaute saadaan koontityökalulta välittömästi muutosten jälkeen. Tätä voidaankin kutsua primitiiviseksi jatkuvan integroinnin muodoksi.

Sbt on mielenkiintoinen ja funktionaalinen koontityökalu. Se on koontityökaluna yksinkertaisempi kuin Maven ja pyrkii ratkaisemaan ohjelmistojen koontiin liittyviä perusongelmia. Sekä Buildr että Gradle tarjoavat enemmän ominaisuuksia Java-projektien koontiin ja molemmat soveltuvat paremmin yleiskäyttöisiksi koontityökaluiksi. Sbt ei ole siis aivan yhtä kunnianhimoinen projekti kuin esimerkiksi Gradle. Sbt on kenties parhaiten Scalaa tukeva koontityökalu ja sen käyttämistä voi suositella Scala-projektien kehittäjille ja niille, jotka haluavat ohjelmoida koontilogiikkaa käyttämällä Scalaa.

3.8 Yhteenveto koontityökaluista

Esitellyistä koontityökaluista Buildr, Gradle ja Sbt ovat ottaneet lähtökohdaksi Mavenin periaatteen, jossa yleisesti käytetyt käytännöt minimoivat konfiguraation tarpeen. Työkalujen kehityksessä on kuitenkin otettu vakavasti Mavenin saama kritiikki sen jäykkyydestä. Listatut kolme työkalua käyttävät suosittuihin ja Turing-vahvoihin ohjelmointikieliin pohjautuvia sovellusaluekohtaisia kieliä. Työkaluja voi helposti laajentaa näillä kielillä ja liitännäisten liittäminen koonnin elämänkaareen on huomattavasti helpompaa kuin Mavenin projektioliomalleissa. Ruby, Groovy ja Scala ovat kaikki päteviä ohjelmointikieliä, joilla on suuret käyttäjäkunnat. Työkalun käyttämä ohjelmointikieli vaikuttaa koontityökalun valintaan, koska prosessikuvauksia on helpointa ohjelmoida itselleen tutulla kielellä. Tämän takia luonnollisesti myös kehitettävän sovelluksen käyttämä kieli ja sovelluskehykset vaikuttavat työkalun valintaan. Sbt on erittäin hyvä valinta Scala-kehitykseen, vaikka työkalulla voi tehdä helpohkosti myös Java-koontia. Buildr on hieman poikkeuksellinen työkalu, sillä sen sovellusaluekohtainen kieli perustuu Rubyyn, vaikka työkalu on ensisijaisesti tarkoitettu Java-ekosysteemin koontityökaluksi.

Todellisia Antin ja Mavenin korvaajia ovat Buildr ja Gradle. Molemmat tarjoavat erittäin laajan ominaisuuspaletin, ja niihin siirtyminen Antista tai Mavenista on melko helppoa. Sekä Buildr että Gradle pystyvät helposti käyttämään Antin tehtäviä ja riippuvuuksienhallinnassa voi hyödyntää Maven- tai Ivy-tietovarastoja. Buildr otettiin vertailuun mukaan sen suosion vuoksi, vaikka sille ei ole Jenkinsissä liitännäistä. Tämän lisäksi Buildr perustuu Rakeen, joka on Rubya käyttävä koontityökalu. Ruby on sinänsä mainio kieli koontiskriptien ohjelmointiin, mutta Metropolian ohjelmistotuotantoprojekteista pääosa on Java-pohjaisia. Java on myös käytetyin kieli opetuksessa ja tuttu kaikille opiskelijoille, joten Gradlen käyttämä Groovy on erittäin helppo omaksua. Gradle on myös ominaisuuksiltaan hieman kypsempi tuote. Näistä syistä Buildria ei voida ottaa käyttöön Metropolian jatkuvan integroinnin ympäristössä.

Metropolian ohjelmistotuotantoprojektissa ryhmät käyttävät hyvin erilaisia ohjelmistoteknologioita ja ohjelmointikieliä, joten yhdellä koontityökalulla ei välttämättä saada katettua kaikkia tarpeita. Gradlella saadaan katettua melko hyvin vähintään kaikki Javaan liittyvät teknologiat. Sillä on myös helppo kirjoittaa yleispäteviä koontitehtäviä, jotka eivät ole tiukasti sidottu mihinkään tiettyyn ympäristöön. Huomioon täytyy kuitenkin ottaa myös tuki integroiduissa kehitysympäristöissä. Gradlellä, kuten muilla

haastajilla, on vielä pitkä matka Mavenin näennäiseen helppokäyttöisyyteen kehitysympäristöjen sisältä.

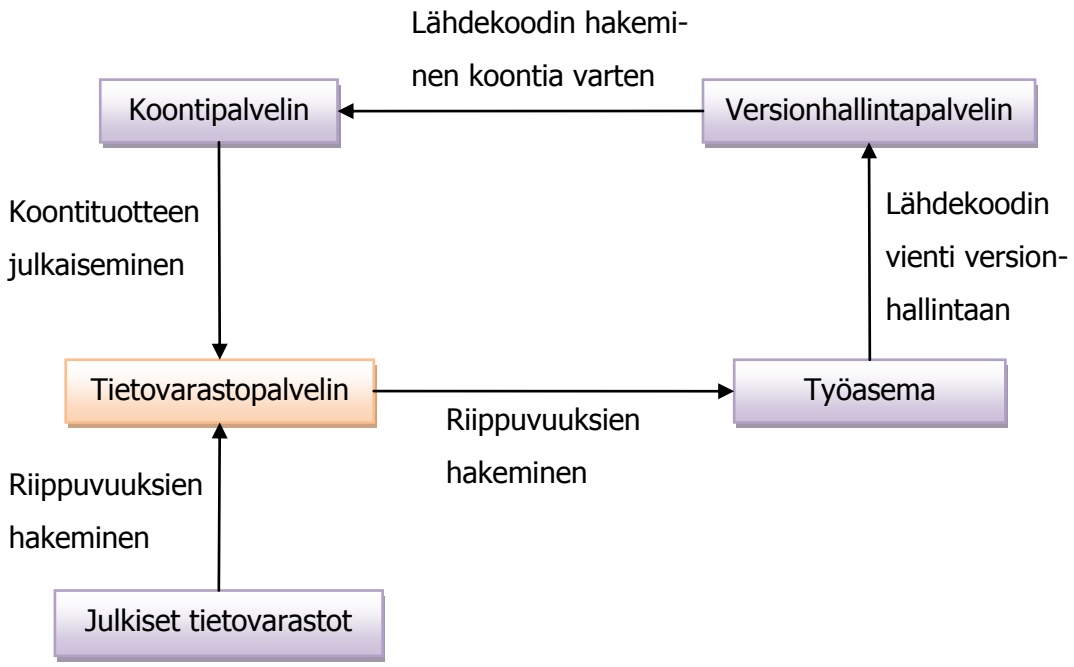
4 Tietovarastonhallintasovellukset

Useimmat ohjelmistoprojektit riippuvat ulkoisista sovelluskehyksistä, kirjastoista tai liitännäisistä. Java-projekteissa riippuvuudet ovat tyypillisesti jar-pakkauksia. PHP:n vastaava formaatti on phar. Riippuvuuksienhallintaan on monia ratkaisuja, joista useat ovat hyvin työläitä. Ulkoisten riippuvuuksien täytyy olla kaikkien projektin kehittäjien saatavilla, ja projekti on optimitilanteessa suoraan käännettävissä, kun se haetaan versionhallinnasta. Komponenttien varastoimiseen voi käyttää esimerkiksi versionhallintaa tai jaettua hakemistoa suoraan tiedostojärjestelmässä. Nämä ratkaisut eivät ole vääriä, mutta ne eivät ole projektin ylläpidettävyyden kannalta hyviä. Modernit hajautetut versionhallintajärjestelmät, kuten Git ja Mercurial, soveltuvat huonosti binääritiedostojen varastoimiseen, koska tietovarastojen kloonaukseen kuuluu täydellisen historian kopiointi [Dockter & Murdoch 2012a; Sadogursky 2012]. Tästä seuraa kloonauksen hidastuminen, koska versioidut tietovarastot kasvavat hyvin suuriksi, mikäli niihin tallennetaan binääritiedostoja. Käsin tehtävä riippuvuuksienhallinta on hankalaa, koska riippuvuuksien versioita vaihdetaan kehityksen aikana ja kehittäjät tarvitsevat uusia riippuvuuksia omille vastuualueilleen. Pahimmassa tapauksessa päädytään lähettämään esimerkiksi jar-pakkauksia sähköpostilla. Tällöin kehittäjillä on täysi työ pitää kirjaa riippuvuuksista, joilla sovellus toimii oikein. Modernien koontityökalujen kehittäjät ovat tunnistaneet riippuvuuksienhallinnan ongelman ja useissa koontityökaluissa riippuvuuksienhallinta on integroitu koontiprosessiin. Näissä työkaluissa riippuvuudet määritellään prosessikuvauksessa ja ne haetaan automaattisesti koontituotteiden varastoimiseen erikoistuneista tietovarastoista yhtenä koonnin vaiheena.

Riippuvuuksienhallinnan automatisointi on nykyisin Java-pohjaisissa projekteissa arkipäivää. Monet avoimen lähdekoodin projektit julkaisevat koontituotteensa julkisiin tietovarastoihin. Kun riippuvuuksienhallintaa hyödyntävää projektia kehitetään, voidaan tarvittavat kirjastot ja muut riippuvuudet hakea julkisista tietovarastoista. Useimmat koontityökalut käyttävät Mavenin standardoimaa riippuvuuksienhallintaa, koska suurin osa julkisista tietovarastoista käyttää Mavenin ymmärtämää rakennetta. Monet työkalut hakevat riippuvuudet oletusarvoisesti Sonatypen ylläpitämästä Mavenin keskustietova-

rastosta. Mavenin käyttämälle rakenteelle vaihtoehtona on Apache Ivyn tukema rakenne. Maven ei pysty käyttämään Ivy-formaatin tietovarastoja, mutta monet modernit koontityökalut, kuten Gradle ja Sbt, käyttävät riippuvuuksienhallintaan Ivya. Nämä työkalut voivat tarvittaessa hyödyntää sekä Maven- että Ivy-tietovarastoja.

Julkisten tietovarastojen käyttäminen on toimiva ratkaisu tiettyyn pisteeseen asti. Oman tietovaraston ylläpitämisessä on kuitenkin muutamia etuja, ja joissain tilanteissa se on lähes välttämätöntä. Julkisten tietovarastojen ongelma on niiden epäluotettavuus. Sovelluksen koonnin pitää olla mahdollisimman hyvin toistettavissa vielä vuosienkin päästä. Julkiset tietovarastot eivät anna takeita siitä, että niiden tallentamat koontituotteet ovat pysyvästi saatavilla. Kaikkia sovelluksen riippuvuuksia ei välttämättä ole alkuunkaan saatavilla julkisista tietovarastoista. Tällaisen riippuvuuden jakaminen täytyy tehdä käsityönä, esimerkiksi versionhallinnan avulla. Projektin ylläpidettävyyden kannalta paras ratkaisu on kaikkien riippuvuuksien hakeminen yhdestä luotettavasta paikasta. Tietovarastonhallintasovellukset tarjoavat ongelmaan ratkaisun. Kehittäjäryhmän käytössä olevaan tietovarastoon voidaan lisätä käsin ulkoisia riippuvuuksia, joita ei ole saatavilla julkisista tietovarastoista. Tämän jälkeen kaikkiin riippuvuuksiin päästään käsiksi yhtenäisellä tavalla koontityökaluista. Tavallisesti koontityökalu konfiguroidaan käyttämään pelkästään kehittäjäryhmän sisäistä tietovarastonpalvelinta, jolla hallitaan kaikkia riippuvuuksia. Tietovarastonhallintasovellus hakee riippuvuudet ensin julkisista tietovarastoista ja tallentaa ne omaan paikalliseen varastoonsa. Riippuvuuksia ei tarvitse hakea julkisista tietovarastoista jokaisen koonnin yhteydessä, vaan ne saadaan nopeasti omalta palvelimelta, jolla tietovarastonhallintasovellusta ajetaan. Tietovarastopalvelin toimii tällaisessa tapauksessa välityspalvelimena koontityökalun ja julkisten tietovarastojen välillä. Monimutkaisissa koonneissa riippuvuuksien hakeminen voi nopeutua huomattavasti, kun julkisiin tietovarastoihin ei tarvitse olla yhteydessä [Repository Management with Nexus]. Kuva 5 havainnollistaa tietovarastopalvelimen roolia koontiprosessissa.



Kuva 5. Tietovarastopalvelimen rooli koontiprosessissa.

Omalla tietovarastopalvelimella on myös helppoa rajoittaa riippuvuuksienhallintatyökalun toimintaa. Rajoituksia voi tehdä esimerkiksi riippuvuuden käyttämän lisenssin mukaan. Esimerkiksi kaupallisen sovelluksen kehityksessä voi olla tarpeellista estää copyleft-lisenssien alaisten tuotteiden päätyminen sovellukseen. [Repository Management with Maven Repository Managers.]

Tietovarastopalvelin on erityisen hyödyllinen kun kehitetään erillisiä projekteja, joiden välillä on riippuvuuksia. Java-projekteissa on tavallista kehittää yleiskäyttöisiä kirjastoja erillään projektista, joka käyttää niitä. Kun käytetään jatkuvaa integrointia, kirjastoista syntyy tiuhaan tahtiin uusia versioita. Kirjastoa hyödyntävä projekti voidaan konfiguroida hakemaan aina uusin kehitysversio *(snapshot)* kirjastosta. Jatkuvan integroinnin voi implementoida siten, että kirjastosta julkaistaan tietovarastoon uusi kehitysversio jokaisen onnistuneen koonnin jälkeen. Tämänkaltaisessa kehityksessä oma tietovarastopalvelin on tärkeä lisä, koska kirjastojen automaattista jakamista on hankala toteuttaa muilla tavoilla. Omalla tietovarastopalvelimella ja riippuvuuksienhallinnan sisältävällä koontityökalulla toisistaan riippuvien projektien kehitys saadaan automatisoitua hyvin pitkälle.

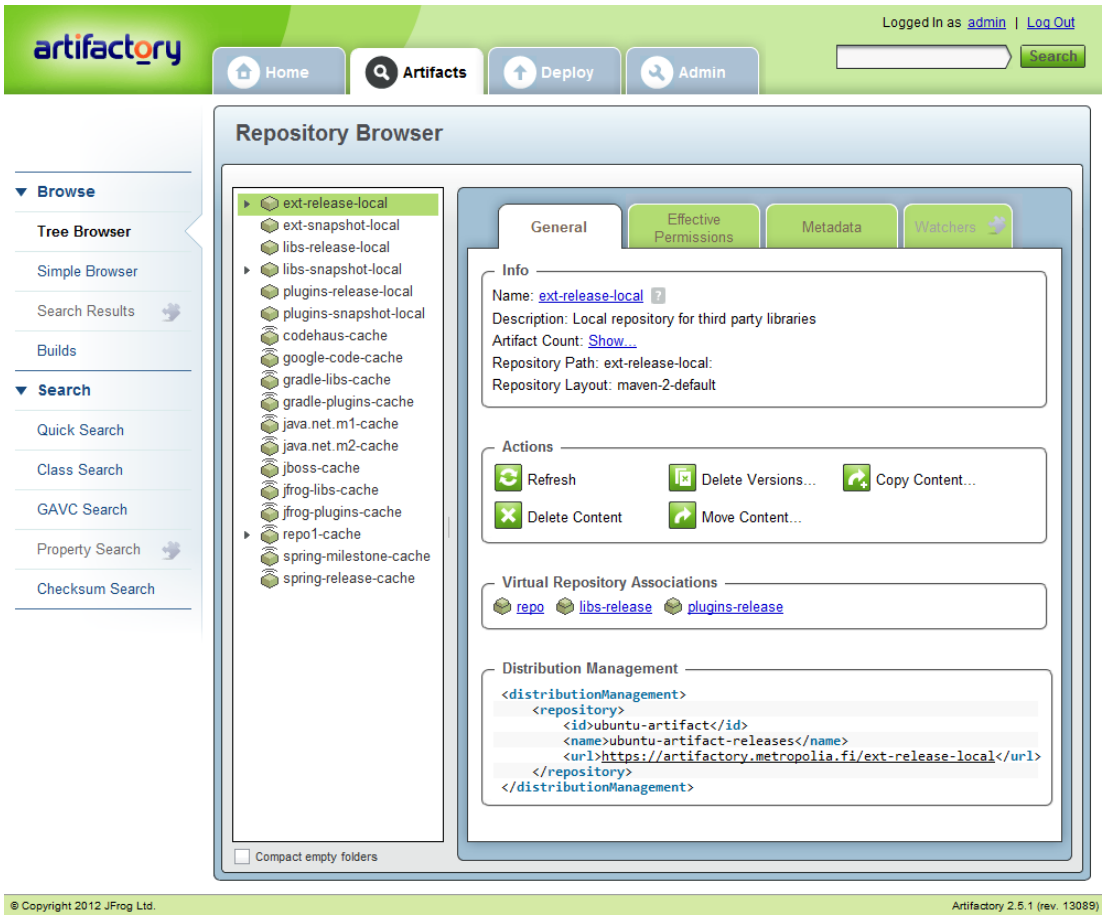
Markkinoilla on muutamia aktiivisesti kehitettäviä tietovarastonhallintasovelluksia: Apache Archiva, JFrog Ltd:n Artifactory Open Source ja Sonatype Nexus. Kaikki kolme sovellusta ovat ulkoisesti hyvin samankaltaisia. Ne ovat Java-sovelluspalvelimella ajettavia web-sovelluksia ja käyttävät oletusjakelussaan Jetty-sovelluspalvelinta. Kaikki kolme toimivat ongelmitta myös suositussa Apache Tomcat -sovelluspalvelimessa. Käyttöliittymiltään sovellukset ovat erinäköisiä, mutta keskeisimmät riippuvuuksienhallintaan liittyvät ominaisuudet löytyvät niistä kaikista. Näistä kolmesta sovelluksesta Artifactory ja Nexus ovat selkeästi suosituimmat ja useimmat verkossa tehdyt vertailut keskittyvät niihin. Archivan suosion vähyyteen on selkeä syy. Archivan kehitys aloitettiin 2007 ja se on sovelluksista uusin. Se ei tarjoa läheskään yhtä kattavaa ominaisuuskirjoa kuin Artifactory ja Nexus [Maven Repository Manager Feature Matrix 2011]. Tässä luvussa on siis keskitytty lähinnä Nexuksen ja Artifactoryn vertailuun.

Sekä Artifactory että Nexus tarjoavat kaikki tietovarastonhallintasovelluksen perusominaisuudet. Molempia voi käyttää välityspalvelimena julkisiin tietovarastoihin ja omia sekä kolmannen osapuolen koontituotteita voi tallentaa sisäisiin tietovarastoihin. Molemmissa sovelluksissa voi hallita useampia tietovarastoja. Kehittäjäryhmät voivat käyttää samaa tietovarastopalvelinta, mutta erillisiä tietovarastoja. Web-käyttöliittymistä voi etsiä riippuvuuksia ja tarkastella niiden riippuvuussuhteita. Molemmat sovellukset tukevat käyttäjähallintaa, jolla voi määritellä ryhmä- ja käyttäjäkohtaisia käyttöoikeuksia. Käyttöoikeuksilla voi rajoittaa pääsyä ja julkaisuoikeuksia tietovarastoihin.

Tyypillinen riippuvuuksienhallintaa käyttävä kehittäjä ohjelmoi Java-ekosysteemin kielellä käyttäen Mavenia koontityökalunaan. Tällaiselle käyttäjälle valinta Artifactoryn ja Nexuksen välillä ei ole helppo, mutta ei myöskään erityisen merkityksellinen. Molemmat tuotteet ovat kypsiä ja toimivat hyvin Mavenin kanssa, eikä niiden paremmuusjärjestyksestä ole selvää konsensusta. Valintaa voi yrittää perustella joillain ominaisuuksilla, ominaisuuksien toteutustavoilla tai esimerkiksi käyttöliittymän käytettävyydellä. Aivan identtisiä sovellukset eivät ole, mutta erot eivät välttämättä käy ilmi tavallisessa käytössä. Molemmat sovellukset ovat ilmaisia ja lisensoitu avoimen lähdekoodin GPL3-lisenssin alle. Molemmista on saatavilla myös kaupalliset versiot, joissa on lisäominaisuuksia. Maksavat asiakkaat saavat luonnollisesti myös teknistä tukea kehittäjäyhtiöltä.

4.1 JFrog Artifactory

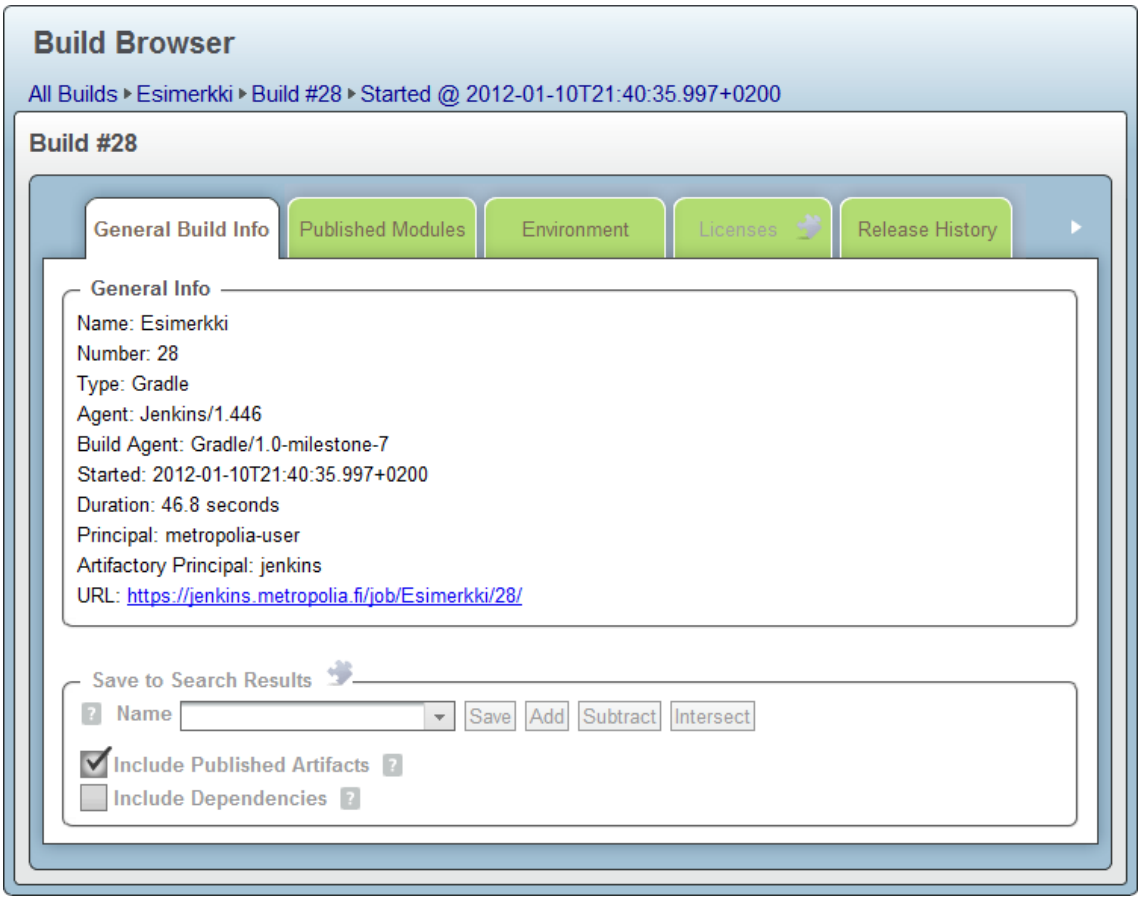
Artifactoryn web-käyttöliittymä on helppokäyttöinen. Ylläpitäjän oikeuksilla käyttöliittymästä voi muokata kaikkia Artifactoryn asetuksia. Ylläpitäjälle rajoitetut asetukset liittyvät tietoturvaan ja käyttäjien ja tietovarastojen hallinnoimiseen. Tiedostojärjestelmän konfiguraatiotiedostoja ei tarvitse muokata missään vaiheessa. Artifactoryn ylläpitäminen on erittäin helppoa ja yksinkertaista. Sovellukseen on sisäänrakennettu jopa varmuuskopiointimekanismi, joka on oletusarvoisesti käytössä. Ylläpitäjä voi muokata varmuuskopioinnin asetuksia ja luoda uusia varmuuskopiointiajoituksia. Tavallisilla käyttöoikeuksilla keskeisimmät käyttöliittymän toiminnot liittyvät koontituotteiden etsimiseen ja lisäämiseen. Käyttöliittymästä voi selata ja etsiä koontituotteita kaikista tietovarastoista, tai haun voi rajoittaa tiettyihin tietovarastoihin. Käyttöliittymästä voi myös generoida Gradlelle, Ivylle ja Mavenille asetustiedostot, joilla koontityökalut konfiguroidaan käyttämään tietovarastopalvelinta. Kuvassa 6 on Artifactoryn web-käyttöliittymä tietovarastonäkymässä.



Kuva 6. Artifactoryn web-käyttöliittymä.

Artifactoryn suurin etu Nexukseen verrattuna on sen jatkuvan integroinnin työkalujen tuki. Artifactoryn tietovarastot voivat käyttää Ivy- tai Maven-tietovarastorakennetta tai täysin räätälöityä rakennetta. Lähes kaikki työkalut pystyvät käyttämään Mavenin ymmärtämää rakennetta, mutta Artifactory soveltuu myös niille käyttäjille, jotka haluavat hyödyntää rakenteessa tunnisteita, joita Maven ei tue. Vapaavalintaisia tunnisteita ovat muun muassa koontituotteen luokitus ja tiedostopääte. Tällaisista tunnisteista koostettuja tietovarastorakenteita voi helposti käyttää Ivylla ja kaikilla koontityökaluilla, jotka käyttävät Ivya riippuvuuksienhallintaan. Artifactory myös integroituu moniin koontityökaluihin. Esimerkiksi Gradlelle on JFrogin kehittämä Artifactory-liitännäinen, jolla koontituotteita voi julkaista suoraan Artifactory-tietovarastoihin. Jatkuvassa integroinnissa kyseistä ominaisuutta ei hyödynnetä, koska koontituotteen julkaisu tietovarastoihin tapahtuu koontipalvelimella.

Yksi mielenkiintoisimmista Artifactoryn ominaisuuksista on integroituminen koontipalvelimien kanssa. Artifactory saa koontipalvelimelta tietoja koontituotteesta, kun se julkaistaan koontipalvelimelta Artifactoryyn. Artifactorystä nähdään muun muassa koontituotteen koonnin järjestysnumero, koontityökalu, koontipalvelin, koonnin ajankohta ja koonnin kesto. Käyttöliittymässä on myös suora linkki koontipalvelimelle, josta nähdään tarkemmat tiedot koonnista. Vastaavasti koontipalvelimella on linkki Artifactoryyn. Artifactoryn maksullisessa Artifactory Pro -liitännäiskokoelmassa on lisäominaisuuksia koontipalvelinintegraatioon, mutta ilmainenkin versio tuo jatkuvaan integrointiin lisäarvoa. Kuvassa 7 on esitetty esimerkki koontipalvelinintegraatiosta ilmaisessa versiossa, jossa aivan kaikki ominaisuudet eivät ole käytössä.



Kuva 7. Artifactoryn koontipalvelin-integraatio.

Artifactoryn koontipalvelinliitännäinen toimii Jenkinsin, Hudsonin, TeamCityn ja Bamboon kanssa. Ominaisuus on yksi suurimmista eroista Nexukseen verrattuna, sillä Nexus ei tue minkäänlaista koontipalvelinintegraatiota.

Artifactory käyttää oletusarvoisesti tietovarastonsa selkärankana Derby-tietokantaa. Nexuksen kehittäjä Brian Fox [2009] on kritisoinut blogikirjoituksessaan Artifactoryn tietokannan käyttöä. Hän mainitsee ongelmiksi erityisesti tietokannan mahdollisen korruptoimisen ja kehuu suoraan tiedostojärjestelmää käyttävää tietovarastoa. Tietojärjestelmään perustuvan varaston eduiksi hän mainitsee varmuuskopioinnin helppouden ja kertoo koontituotteiden tuonnin ja viennin onnistuvan Nexuksessa suoraan tiedostojärjestelmästä käsin. Tietokannan korruptoimisen riski on luultavasti aika minimaalinen, mutta joka tapauksessa tiedostojärjestelmän käyttäminen tietovaraston pohjana voi tarjota etuja joissain käyttötapauksissa. Erityisesti erittäin suurien tietovarastojen tapauksessa tietokantojen suorituskyky voi laskea.

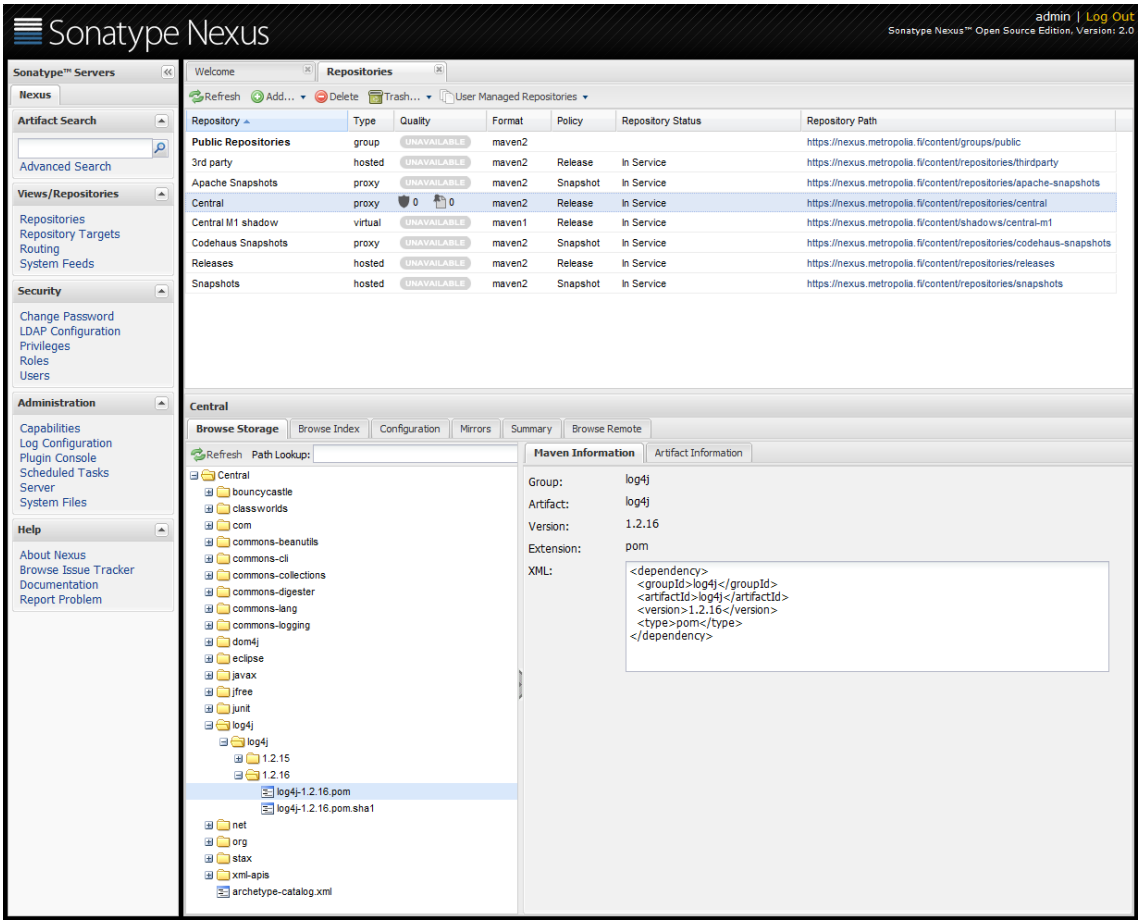
Nykyisin Artifactory voidaankin konfiguroida käyttämään tiedostojärjestelmää varastona kuten Nexus. Jopa koontituotteisiin liittyvät metatiedot voidaan tallentaa suoraan tiedostojärjestelmään, mutta tätä ei suositella. Mikäli käytettävällä palvelinkoneella on valmiiksi asennettuna jokin tietokantapalvelinsovellus, saatetaan sitä haluta käyttää Artifactoryn tietokantapalvelimena. Artifactory voidaankin helposti konfiguroida käyttämään Derby-tietokannan sijaan valmista tietokantainfrastruktuuria, joka perustuu johonkin suosituimmista relaatiotietokantaohjelmistoista. Tuettuja ohjelmistoja ovat muun muassa MySQL, PostgresSQL ja Microsoft SQL Server. [Landman & Simon 2011.]

JFrog tarjoaa Artifactoryyn maksullista Artifactory Pro -liitännäiskokoelmaa. Pakettiin kuuluu monia mielenkiintoisia liitännäisiä, mutta ilmaisessa versiossa on kaikki oleellisimmat ominaisuudet tavallisiin käyttötapauksiin. Artifactory Pro -pakkaus avaa myös mahdollisuuden kehittää omia Groovylla toteutettuja liitännäisiä. Maksulliset liitännäiset lisäävät Artifactoryyn tuen muun muassa LDAP-ryhmille. Tämä on hyödyllinen ominaisuus organisaatioille, joilla on valmis LDAP-infrastruktuuri, johon ryhmien oikeudet on määritelty. Perinteinen LDAP-autentikointi onnistuu myös ilmaisessa versiossa. Maksullisiin liitännäisiin kuuluu myös useiden tietovarastoformaattien tuki. Artifactoryn maksullinen versio tukee muun muassa YUM-, P2- ja NuGet-tietovarastoja. Kenties mielenkiintoisin lisä on tuki NuGet-tietovarastoille. NuGet on riippuvuuksienhallintajärjestelmä .NET-ympäristöön, joka integroituu Visual Studioon.

4.2 Sonatype Nexus

Nexuksen käyttöliittymä on hyvin ammattimaisen näköinen ja tuntuinen. Kaikki toiminnot löytyvät intuitiivisista paikoista ja toiminnot ovat hyvin responsiivisia. Käyttöliittymä hyödyntää hyvin toimivaa välilehtimallia, jonka ansiosta useampia käyttöliittymän osioita voi selata ja muuttaa samanaikaisesti tallentamatta osioiden välillä. Nexuksen käyttöliittymä on ulkoasultaan ja tuntumaltaan hieman Artifactoryn käyttöliittymää edellä. Varsinaisilta toiminnoiltaan molempien sovelluksien käyttöliittymät tarjoavat lähes samat mahdollisuudet. Kuten Artifactoryssä, myös Nexuksessa kaikki asetukset ovat ylläpitäjäoikeuksilla konfiguroitavissa suoraan käyttöliittymästä. Nexuksen hakujärjestelmä eroaa hieman Artifactoryn vastaavasta. Artifactoryn haku mahdollistaa ainoastaan sen sisäisiin tietovarastoihin tallennettujen koontituotteiden hakemisen. Nexus vie haun askelta pidemmälle mahdollistamalla hakujen tekemisen ulkoisiin tietovarastoihin, joi-

den koontituotteita ei ole vielä Nexukseen säilötty. Ominaisuus helpottaa riippuvuuksien löytämistä huomattavasti. Nexuksen web-käyttöliittymä on esitetty kuvassa 8.



Kuva 8. Nexuksen web-käyttöliittymä.

Toisin kuin Artifactory, Nexus ei käytä tietovaraston selkärankana oletusarvoisesti tietokantaa. Sen sijaan koontituotteet tallennetaan suoraan tiedostojärjestelmään. Nexuksen tietovarasto on periaatteessa helpompi varmuuskopioida, koska se onnistuu yksinkertaisella ajastetulla kopioinnilla [Xu 2010]. Artifactory on kuitenkin ratkaissut tietokannan varmuuskopioinnin tarjoamalla sisäänrakennetun varmuuskopiointiominaisuuden, jota on helppo käyttää suoraan käyttöliittymästä. Nexuksessa tällaista ominaisuutta ei ole, joten varmuuskopioinnin käyttöönotto on yksinkertaisuudestaan huolimatta hieman työläämpää kuin Artifactoryssä.

Nexuksen maksullinen Pro-versio lisää ominaisuusrepertuaariin useita ominaisuuksia. Ominaisuudet ovat osittain samoja kuin Artifactoryn maksullisessa versiossa. Nexus Pro

tukee P2- ja NuGet-tietovarastoja. Myös laajempi LDAP-tuki kuuluu maksullisen version ominaisuuksiin. Ominaisuudet ovat sinänsä mielenkiintoisia, mutta edes Nexuksen maksullinen versio ei integroidu koontipalvelinsovelluksiin millään tavalla, vaikka Sonatype kehittää Jenkins CI:n kanssa kilpailevaa Hudsonia. Nexus ei myöskään virallisesti tue mitään muita riippuvuuksienhallintatyökaluja kuin Mavenia.

4.3 Yhteenveto tietovarastonhallintasovelluksista

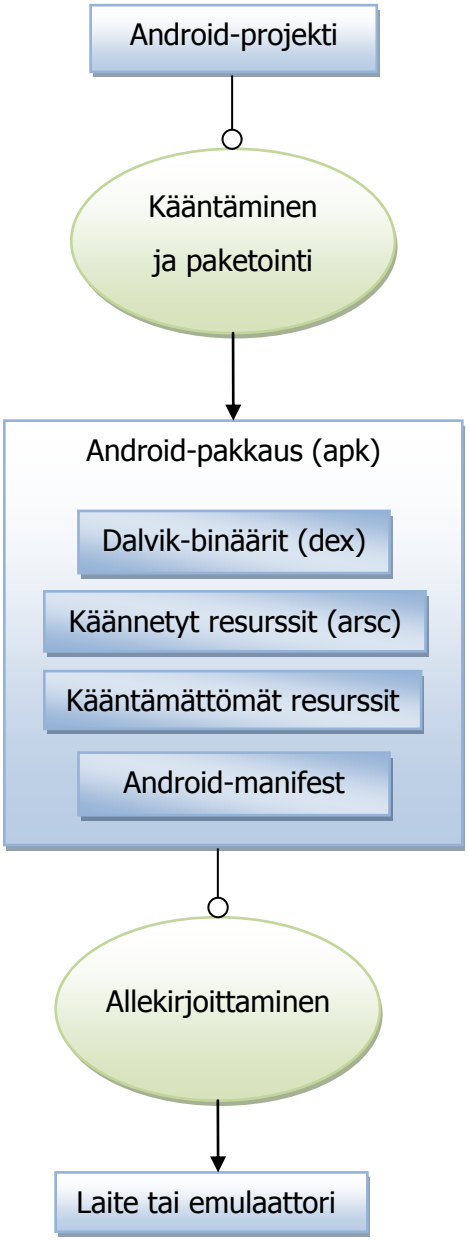
Tietovarastopalvelimen ylläpitäminen helpottaa riippuvuuksienhallintaa jatkuvan integroinnin ympäristössä. Metropolian ohjelmistotuotantoprojekteissa tietovarastopalvelin olisi erityisen hyödyllinen välityspalvelimena. Monet projektit käyttävät samoja tunnettuja avoimen lähdekoodin sovelluskehyksiä ja kirjastoja, joiden lataamisessa kestää pahimmillaan useita minuutteja. Koulun sisäisestä tietovarastosta riippuvuudet löytyisi sekunneissa ja kehitys nopeutuisi. Projektien koontituotteita voitaisiin myös julkaista tietovarastoon. Tietovarastoon tallennettuja koontituotteita on helppo käyttää muissa projekteissa.

Artifactory ja Nexus ovat melko tasavahvoja tietovarastonhallintasovelluksia. Erityisen huomionarvoista on kuitenkin Artifactoryn ja Nexuksen integroituminen muihin jatkuvan integroinnin sovelluksiin. Nexus on tiukasti sidottu Sonatypen kehittämään koontityökaluun, Maveniin. Lisäksi Sonatype kehittää yhteistyössä Oraclen kanssa Hudson-koontipalvelinsovellusta, jonka kehityksessä on panostettu erityisesti hyvään Maven-integraatioon. Nexuksen valinta täytyy siis tehdä harkiten ja tiedostaen kehittäjäyhtiön tavoitteet. Sonatypen intresseissä on tarjota mahdollisimman hyvin yhteen nivoutuva tuoteperhe. Kilpailevien työkalujen integroiminen heidän ohjelmistoihin ei todennäköisesti ole kovin korkealla prioriteettilistalla. Artifactory integroituu hyvin Mavenin lisäksi myös muiden riippuvuuksienhallintatyökalujen kanssa. Lisäksi Artifactory tukee useita erilaisia koontipalvelinsovelluksia. Tämän ansiosta Artifactory taipuu huomattavasti useampaan käyttötapaukseen kuin Nexus. [Ben Haim 2011.] Metropolian ohjelmistotuotantoprojektissa eri ryhmät käyttävät erilaisia työkaluja riippuen muun muassa ryhmän omista tarpeista ja asiakkaan vaatimuksista. Jatkuvan integroinnin työkaluja täytyy siis tukea ympäristössä mahdollisimman laaja-alaisesti.

Osa Nexuksen ominaisuuksista on selvästi Artifactorya kehittyneempiä, mutta sen rajoitteet tekevät siitä huonon valinnan Metropolian ohjelmistotuotantoprojektien kehitysympäristöön. Tietovarastopalvelinta ei vielä tämän opinnäytetyön tekemisen aikana liitetty ympäristöön, mutta Artifactoryn käyttöönotto on virtualisoidussa palvelinympäristössä erittäin helppoa. Artifactoryn generoimat asetustiedostot tekevät käyttöönotosta helpon myös riippuvuuksienhallintatyökaluissa.

5 Android-projektin koonti

Android on Open Handset Alliancen kehittämä mobiilikäyttöjärjestelmä. Open Handset Alliancea johtaa Google. Android on noussut nopeasti suosituksi kehitysalustaksi. Android-sovellusten koonti muistuttaa Java-sovellusten koontia, sillä sovellukset ohjelmoidaan Javalla. Java-koodin staattiseen analyysiin tarkoitetut sovellukset käyvät sellaisenaan Android-projektien laaduntarkastukseen. Monille työkaluille on lisäksi Android-spesifisiä sääntöjä, jotka havaitsevat erityisesti Android-koodiin liittyviä ongelmia [PMD - Android Rules]. Android-sovellusten koontiin liittyy huomattavasti enemmän välivaiheita ja työkaluja, kuin perinteiseen Java-koontiin. Myös lopullinen laitteeseen tai emulaattoriin asennettava apk-pakkaus eroaa merkittävästi jar-pakkauksista. Kehityksen aikana tuotetut Android-resurssit ja lähdekoodi muovataan Java-kääntäjän ymmärtämään muotoon, ja käännöksen jälkeen luokkatiedostot muunnetaan Androidin Dalvik-virtuaalikoneen ymmärtämään dex-formaattiin. Lopuksi kaikki resurssit paketoidaan apk-pakkaukseksi, joka allekirjoitetaan. Koontiprosessi on esitetty yksinkertaistettuna kuvassa 9.



Kuva 9. Android-projektin koontiprosessi [Building and Running].

Android-koonti valittiin tutkimustapaukseksi, koska Metropolian ohjelmistotuontanto-projekteissa on kehitetty useita Android-sovelluksia, joiden kehityksessä ei ole hyödynnetty jatkuvaa integrointia. Koonti onnistuu muutamilla Jenkins CI:n tukemilla koontityökaluilla, joten sen voi integroida osaksi Metropolian kehitysympäristöä. Koontia kunnolla tukevia työkaluja ei ole kovin montaa, koska Android on suhteellisen uusi kehitysalusta ja koontiprosessi on melko monimutkainen. Jokaiseen kuvan 9 vaiheeseen liittyy erillisiä työkaluja, jotka kuuluvat Android SDK:iin.

Android-kehitystyökaluja on helpointa käyttää Eclipsen kautta. Työkalut integroituvat Eclipseen Android Developer Tools -liitännäisellä. ADT helpottaa kehitystä huomattavasti, sillä se integroi kaikki kehityksen vaiheet saumattomasti Eclipseen. Liitännäinen lisää Eclipseen Android-projektipohjan, käyttöliittymäeditorin, debuggerin ja muita ominaisuuksia [ADT Plugin for Eclipse]. Sovelluksen voi jopa lähettää testattavaksi oikeaan laitteeseen suoraan Eclipsen sisältä. Kehitystä varten ei tarvitse konfiguroida koontityökalua, vaan kaikki tarvittavat asetukset voi tehdä graafisesti Eclipsellä.

Eclipsen lisäksi Android-kehitys onnistuu esimerkiksi NetBeans- ja IntelliJ IDEA-ohjelmointiympäristöissä. Kaikki kolme suosituinta Java-kehitysympäristöä ovat siis varteenotettavia vaihtoehtoja varsinaiseen kehitystyöhön. Pelkän kehitysympäristön tuki ei kuitenkaan riitä jatkuvan integroinnin käyttöönottoon. Sovelluksen koonti täytyy erottaa kehitysympäristöstä erillisen koontityökalun tehtäväksi. Kun koonti pystytään paikallisesti suorittamaan koontityökalulla, onnistuu se helposti myös koontipalvelimella. Jenkins CI tukee Android-koontia hyvin; koontiprosessi ei Jenkinsin kannalta eroa juurikaan Java-koonnista. Jenkinsiltä vaaditaan kuitenkin tuki Android-emulaattorin käynnistämiseen koonnin aikana. Androidin kehitystyökaluihin kuuluu JUnit-laajennos, jonka avulla sovelluksen ajonaikaisen testauksen voi automatisoida. Niin kutsutuilla instrumentointitesteillä pystytään toteuttamaan funktionaalisia testejä, jotka simuloivat käyttäjän tekemiä syötteitä emulaattorilla ajettavaan sovellukseen. Tällaisten testien suorittaminen onnistuu koontipalvelimella, sillä Jenkinsiin on saatavilla laaja Android-emulaattorin halutuilla parametreillä käynnistävä liitännäinen [Orr 2012].

Androidin kehitystyökaluilla voi generoida Ant-prosessikuvauksen projekteille. Generoitu prosessikuvaus sisältää kuvassa 9 esitettyjen vaiheiden lisäksi tehtävät testien ajamiseen ja testikattavuuden määrittelyyn Emmalla. Muita laaduntarkastukseen liittyviä tehtäviä prosessikuvauksessa ei ole, joten niiden lisäämiseksi prosessikuvausta täytyy muokata käsin. Prosessikuvaus on yli 1000 riviä pitkä, eikä sen muokkaaminen ole aivan helppoa. Ant on silti suhteellisen helppo ottaa käyttöön Jenkinsin kanssa, ja Antia voi pitää ensisijaisena valintana Android-projektien koontityökaluksi. Antin lisäksi Android-koonti onnistuu myös Gradlellä ja Mavenilla. Tässä luvussa tarkastellaan näiden kolmen koontityökalun käyttöä Android-projektien koonnissa.

5.1 Gradlen Android-liitännäinen

Android SDK:n työkalut sisältävät komentorivisovelluksen, jolla voi generoida Android-projektin rungon. Komento projektin luontiin on esitetty koodiesimerkissä 15.

```
android create project --target 1 --path ./AndroidDemo --activity DemoActivity
--package fi.metropolia.androiddemo
```

Koodiesimerkki 15. Android-projektin luonti komentorivisovelluksella.

Projektin voi luoda helposti myös Eclipsen ADT-liitännäisellä, mutta komentoriviltä generoitu projekti on paras pohja erillisen koontityökalun käyttöönottoon. Generoidun projektin koonti onnistuu Gradlen Android-liitännäisen avulla helposti. Projektin hakemistorakenteeseen tai kehitystyökalujen luomiin asetustiedostoihin ei tarvitse koskea. Projektin koontiin tarvitaan vain Gradlen prosessikuvaustiedosto, joka käyttää Android-liitännäistä. Koodiesimerkissä 16 esitetty prosessikuvaus tarjoaa tavalliset Android-koontiin tarvittavat tehtävät.

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'org.gradle.api.plugins:gradle-android-plugin:1.1.0'
    }
}

apply plugin: 'android'
apply plugin: 'findbugs'

sourceCompatibility = 1.6

repositories {
    mavenCentral()
}

sourceSets {
    main {
        java {
            srcDir 'src'
        }
    }
}

androidSignAndAlign {
    keyStore = "my-release-key.keystore"
    keyAlias = "demo"
    keyStorePassword = "salasana"
    keyAliasPassword = "salasana"
}
```

```
task configureDebug << {
    jar.classifier = "debug"
}

task configureRelease << {
    proguard.enabled = true
}

androidEmulatorStart {
    avdName = "Android-2.1"
}
```

Koodiesimerkki 16. Gradle-prosessikuvaus Android-sovelluksen koontiin.

Yksinkertaisuudestaan huolimatta prosessikuvauksen tehtävät riittävät hyvin pitkälle. Prosessikuvausta käyttämällä sovelluksen voi allekirjoittaa joko erityisellä debug-avaimella tai kehittäjän omalla yksityisellä avaimella. Debug-avainta käytetään sovelluksen kehitysvaiheessa ja julkaistava versio allekirjoitetaan yksityisellä avaimella. Prosessikuvauksessa on myös määritelty koonnin yhteydessä käynnistettävän emulaattorin nimi. Lisäksi prosessikuvaus käyttää Gradleen sisäänrakennettua FindBugs-liitännäistä. FindBugs on usein Java-projekteissa käytetty laaduntarkastustyökalu. Luotu projekti ei noudata Mavenin hakemistomallia, joten lähdekoodien hakemisto täytyy määritellä prosessikuvaksessa. Vaihtoehtoisesti projektin rakenne voitaisiin muuttaa Mavenin standardimallin mukaiseksi.

Android-sovellusten koonti on melko monimutkainen ja virheherkkä prosessi. Kuva 9 esittää asian yksinkertaistetusti, mutta jokaiseen vaiheeseen liittyy monia työkaluja. Tämä näkyy myös Ant-prosessikuvauksen pituudessa. Kirjoitushetkellä Java Development Kit 7:ään kuuluva jarsigner-työkalu ei toiminut Gradle-liitännäisen kanssa, vaan koonti onnistui täydellisesti ainoastaan JDK 6:lla. Ant-pohjaisessa koonnissa ongelma oli jo ratkaistu, mutta Maven-liitännäistä vaivasi sama ongelma. Yhteensopivuusongelmia syntyy helposti, kun Android-kehitystyökaluista julkaistaan uusia versioita. Ant on virallisesti tuettu koontityökalu, joten se on hieman muita koontityökaluja luotettavampi vaihtoehto Android-koontiin. Kolmannen osapuolen liitännäisten päivitystahti on vaihtelevaa, koska projekteilla ei ole takanaan suurta kehittäjäorganisaatiota.

Gradlen etu Antiin ja Maveniin verrattuna on sen prosessikuvauksen helppo muokattavuus. Laaduntarkastustyökalujen liittäminen koontiprosessiin on helppoa, koska koonnin yksityiskohdat on piilotettu liitännäisen sovelluslogiikan taakse. Usein käytettyjä Java-laaduntarkastustyökaluja, kuten Findbugsia ja PMD:tä, voi helposti käyttää And-

roid-koonnissa. Niiden käyttöönotto ei eroa mitenkään Java-koonnista. Toisaalta generoitu Ant-prosessikuvaus sisältää tehtävät myös Android SDK:n työkalujen kutsumiseen, mikä on hyödyllistä, jos koontiprosessin sisuksia halutaan muokata. Valtaosalla käyttäjistä ei kuitenkaan ole mitään tarvetta muokata Android-koonnin ydintehtäviä, joten hyöty jää kyseenalaiseksi. Gradlen prosessikuvaus on huomattavasti yksinkertaisempi ja luettavampi kuin Ant-prosessikuvaus. Tämä johtuu osittain liitännäisen abstrahoimasta koontilogiikasta, mutta myös Gradlen Groovy-pohjaisesta sovellusaluekohtaisesta kielestä. Gradlen prosessikuvaukset ovat hyvin tiiviitä ja helposti ymmärrettäviä. Antin ja Mavenin XML-muotoiset prosessikuvaukset ovat selvästi monisanaisempia ja vaikealukuisempia.

5.2 Mavenin Android-liitännäinen

Mavenin Android-liitännäinen muistuttaa hyvin paljon Gradle-liitännäistä. Gradlen ja Mavenin perustavanlaatuiset erot ovat suuria, mutta molempien työkalujen Android-liitännäiset toimivat hyvin pitkälti samoilla periaatteilla. Mavenin Android-liitännäinen on helpointa ottaa käyttöön generoimalla projekti koodiesimerkin 15 komennolla ja lisäämällä projektin juureen prosessikuvaustiedosto. Koodiesimerkki 17 havainnollistaa Android-koonnin projektioliomallia.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-
instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.o
rg/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
    <groupId>fi.metropolia.androidesimerkki</groupId>
    <artifactId>androidesimerkki</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>apk</packaging>
    <name>esimerkki</name>

    <dependencies>
        <dependency>
            <groupId>com.google.android</groupId>
            <artifactId>android</artifactId>
            <version>1.6_r2</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    <build>
        <finalName>${project.artifactId}</finalName>
        <sourceDirectory>src</sourceDirectory>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>com.jayway.maven.plugins.android.generation2</groupId>
```

```
                    <artifactId>android-maven-plugin</artifactId>
                    <version>3.1.1</version>
                    <extensions>true</extensions>
                </plugin>
            </plugins>
        </pluginManagement>
        <plugins>
            <plugin>
                <groupId>com.jayway.maven.plugins.android.generation2</groupId>
                <artifactId>android-maven-plugin</artifactId>
                <configuration>
                    <sdk>
                        <platform>7</platform>
                    </sdk>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Koodiesimerkki 17. Minimaalinen projektioliomalli Android-koontiin.

Esitetty projektioliomalli on mahdollisimman yksinkertainen. Koodiesimerkissä 16 esitetty Gradle-prosessikuvaus tarjoaa laajemmat toiminnallisuudet koontiin. Mavenin projektioliomallin monisanainen syntaksi vaatisi samojen koonnin päämäärien *(goal)* lisäämiseksi runsaasti lisäkuvauksia. Periaate päämäärien lisäämiseen on kuitenkin sama kuin Gradle-pohjaisessa koonnissa. Koodiesimerkin 17 projektioliomalli ei sisällä päämääriä julkaisukelpoisen apk-pakkauksen allekirjoittamiseen eikä emulaattorin käynnistämiseen. Mavenilla toteutettuun Androd-koontiin voi helposti liittää laaduntarkastustyökaluja.

5.3 Yhteenveto Android-koonnista

Android-koonti on koontityökalujen kannalta lähinnä Java-koonnin erikoistapaus. Ant on helppo valinta koontityökaluksi, koska Googlen kehittämät työkalut tukevat sitä. Koontityökalua valitessa täytyy Android-tuen lisäksi ottaa huomioon työkalujen perusominaisuudet. Gradle ja Maven tarjoavat sisäänrakennetun riippuvuuksienhallinnan, joka on erittäin hyödyllinen Android-projekteissa. Riippuvuuksienhallinta toimii samalla tavalla kuin Java-projekteissa, ja Android-projektit voivat käyttää samoja komponentteja kuin Java-projektit. Ant ei sisällä riippuvuuksienhallintaa eikä Android-kehitystyökalut tue suoraan esimerkiksi Ivya.

Gradlen ja Mavenin erot Android-koonnin suhteen ovat olemattomia. Valinta näiden kahden työkalun välillä täytyy tehdä muiden ominaisuuksien perusteella. Gradle on selvästi modernimpi työkalu ja sen prosessikuvaukset pysyvät monimutkaisissakin koonneissa melko yksinkertaisina. Maven on vastaavasti paremmin tuettu kehitysympäristöissä. Esimerkiksi Mavenin Eclipseen integroiva m2eclipse-liitännäinen toimii myös Android-koonnin kanssa. Jos kehitystyötä tehdään Eclipsellä ja Mavenilla, voi koonnin suorittaa helposti Eclipsen sisältä. Gradlen integroituminen kehitysympäristöihin on melko alkeellisella tasolla, joten Gradlen Android-koonti vaatii komentorivin käyttöä.

Jenkinsin kannalta Android-koontityökalun valinnalla ei ole merkitystä. Ant, Gradle ja Maven ovat kaikki hyvin tuettuja Jenkinsissä ja koonti tapahtuu samoilla mekanismeilla. Jenkinsin emulaattoriliitännäinen ei ole riippuvainen käytetystä koontityökalusta. Koontityökalun valinta Android-koontiin on melko haasteellista. Yksikään kolmesta tutkitusta työkalusta ei ole selvästi ylitse muiden, joten työkalu täytyy valita organisaation tai projektin tarpeiden mukaan. Ant on sopiva työkalu useimpiin Metropolian ohjelmistotuotantoprojekteihin, mutta automatisoitua riippuvuuksienhallintaa kaipaavat projektit hyötyvät Gradlestä tai Mavenista.

6 PHP-projektin koonti

Jatkuvan integroinnin hyödyntäminen ei ole levinnyt PHP-kehittäjien keskuudessa yhtä laajalle kuin Java-maailmassa. Tämä selviää helposti tarkastelemalla jatkuvan integroinnin työkalutarjontaa. Koontityökalut ja koontipalvelinsovellukset ovat hyvin keskittyneitä Javaan. Monet PHP-kehittäjät ovat tottuneet toimintatapaan, jossa paikallisella työasemalla tehtävän kehityksen jälkeen muutokset yksinkertaisesti lähetetään testipalvelimelle ja testataan sovelluksen toimivuus käymällä testitapaukset käsin läpi. Tässä luvussa tarkastellaan yksittäisiä PHP-sovelluksen laaduntarkastukseen soveltuvia työkaluja ja tutkitaan koontityökalujen mahdollisuuksia niiden hyödyntämiseen.

PHP-projektin koontiin voi käyttää useita koontityökaluja. Kuten luvussa 3.6 todettiin, Phing on PHP-koontiin erikoistunut työkalu. Phing on erityisen varteenotettava vaihtoehto tapauksissa, joissa PHP-sovelluksen koontiin tarvitaan monimutkaista koontilogiikkaa ja PHP on ohjelmointikielenä kehittäjien vahvin osaamisalue. Phingin liitännäiset ohjelmoidaan PHP:lla, joten PHP-kehittäjät voivat helposti tehdä liitännäisiä projektien-

sa tarpeisiin. Kun tämä ominaisuus jätetään huomioimatta, vähenee Phingin edut Antiin verrattuna olemattomiin. Metropolian ohjelmistotuotantoprojekteissa sovellusta varten räätälöidyn koontilogiikan ohjelmointi ei yleensä ole tarpeen, joten Phingin käyttöä on vaikea perustella. Kun yhtälöön otetaan mukaan Jenkins CI -koontipalvelin, Ant osoittautuu selkeästi paremmaksi vaihtoehdoksi. Tämä johtuu parhaiten PHPUnitin kehittäjänä tunnetun Sebastian Bergmannin työstä PHP:hen liittyvien jatkuvan integroinnin työkalujen parissa. Bergmann pitää henkilökohtaisesti Antista ja on kehittänyt useita työkaluja, jotka helpottavat Antin käyttöä Jenkinsin kanssa [Bergmann 2011a]. Antin käyttämistä ja siihen liittyviä Bergmannin työkaluja käsitellään tarkemmin luvussa 6.2. Yksittäisiä laaduntarkastustyökaluja käsitellään luvussa 6.1.

Antin ja Phingin prosessikuvauksien kieli on imperatiivista. Java-maailmassa Maven ja useimmat modernit koontityökalut ovat ottaneet prosessikuvauksien lähtökohdaksi deklaratiivisuuden. Prosessikuvauksien deklaratiivisuutta pidetään nykyisin hyvänä asiana, joskin monilla moderneilla koontityökaluilla koontilogiikkaa voi ohjelmoida myös imperatiivisesti. Vaikka Antin ja Phingin prosessikuvauskielet ovat imperatiivisia, mikään ei estä PHP-koonnin deklaratiivista kuvausta. Ongelmana on sopivan koontityökalun puute. Mavenille on kehitetty kunnianhimoinen PHP-koontiin tarkoitettu liitännäinen, Maven for PHP. Liitännäinen pyrkii tuomaan Mavenin PHP-kehittäjien ulottuville. Maven for PHP tarjoaa riippuvuuksienhallinnan ja deklaratiivisen tavan kuvata PHP-projektin koontia. Koonti tapahtuu edelleen Java-koonnista tutun elämänkaarimallin mukaisesti. Liitännäinen on hyvin uskollinen Mavenin alkuperäiselle ideologialle ja Mavenin hyvät ja huonot puolet pätevät myös siihen. PHP-sovellusten koontiin liittyy kuitenkin haasteita, joita on hankala Mavenin kaltaisella koontityökalulla ratkaista. Liitännäistä käsitellään luvussa 6.3.

Luvussa 3.5 esitelty Gradle on erittäin hyvä yleiskäyttöinen koontityökalu. Gradlen kehityksessä keskitytään ensisijaisesti Java-ekosysteemiin, eikä sille ole olemassa Maven for PHP:n kaltaista liitännäistä. Liitännäisten kehittäminen Gradlelle on sen tehokkaan rajapinnan ansiosta melko helppoa. Luvussa 6.4 esitetään yksinkertaisen PHP-laaduntarkastusliitännäisen toteutus. Liitännäisen kehityksen tarkoituksena oli tutkia Gradlen soveltuvuutta PHP-koontiin. Toteutus pidettiin mahdollisimman yksinkertaisena, eikä siihen liitetty esimerkiksi Maven for PHP:n riippuvuuksienhallintamekanismia.

6.1 PHP:n laaduntarkastustyökalut

PHP on dynaamisesti tyypitetty ja tulkattava ohjelmointikieli, joten ohjelmointivirheet havaitaan yleensä vasta ajon aikana. Ajon aikana virheet havaitaan esimerkiksi käymällä läpi testitapauksia selaimella. Tämä voi tapahtua käsin tai automatisoidusti. Käsityötä halutaan kuitenkin laaduntarkastuksessa välttää. Sovellukset muuttuvat nopeasti kehityksen aikana, eikä käsin tehtäviä testejä ole aikaa suorittaa jokaisen muutoksen jälkeen. Ongelmien havaitseminen ja korjaaminen mahdollisimman aikaisin on ohjelman laadun ja ylläpidettävyyden kannalta tärkeää. Konsepti on hyvin keskeinen jatkuvassa integroinnissa. Kunnollisia laaduntarkastusmekanismeja hyödyntämällä PHP ei ole ohjelmakoodin laaduntarkastuksen suhteen huonommassa asemassa kuin esimerkiksi Java. Tässä luvussa esiteltävät työkalut ovat kaikki komentorivisovelluksia, joita voi helpohkosti kutsua koontityökaluista. Kaikki esiteltävät työkalut voi konfiguroida komentoriviparametreilla, mutta monet niistä tukevat myös XML-konfiguraatiotiedostoja.

Tärkein työkalu PHP:n laaduntarkastukseen on PHPUnit. Nimensä mukaisesti PHPUnit on sovelluskehys yksikkötestien tekemiseen. PHPUnitilla voi toteuttaa myös integrointitestejä, joissa testataan oikeaa tietokantaa tai sovelluksen muita ulkoisia riippuvuuksia vasten. PHPUnitin inspiraationa on ollut JUnit, joka on Java-maailman suosituin testikehys [Bergmann 2006: 1]. JUnitia käyttäneen kehittäjän on helppo omaksua PHPUnitin toimintatavat. PHPUnitin XML-muotoinen raportti noudattaa täysin samaa formaattia kuin JUnit, joten sitä on helppo käyttää raportteja käsittelevien sovellusten, kuten Jenkins CI:n kanssa. PHPUnit tukee asetuksien määrittämistä sekä komentoriviparametreina että XML-konfiguraatitiedostossa. Esimerkki konfiguraatiotiedostosta on liitteen 2 koodiesimerkissä 3.

Monet staattisen analyysin työkaluista liittyvät mittareihin, joilla projektin tilaa voi tutkia määrällisesti. Tällaiset ohjelmistomittarit ovat hyödyllisiä laadun analysointiin, mutta niitä saatetaan käyttää myös esimerkiksi kustannusarvioiden tekemiseen. PHP_Depend generoi PHP-koodin pohjalta mittareita, joilla koodin laatua voi analysoida. PHP_Depend on saanut vaikutteita JDependistä ja NDependistä. Ne ovat nimiensä mukaisesti vastaavanlaisia staattisen analyysin työkaluja Java- ja .NET-ympäristöihin. Mittarit ovat erityisen hyödyllisiä kun kartoitetaan refaktoroinnin tarpeessa olevia ohjelman osioita. Mittareita ovat muun muassa syklomaattinen kompleksisuus, projektihierarkian syvyys, kommenttirivien määrä koodirivejä kohti ja luokkien koko. Mittareiden lukemia

ja muutoksia analysoimalla voi havaita rakenteellisia ongelmia, jotka voivat hankaloittaa projektin jatkokehitystä ja ylläpitoa. PHPLOC on PHP_Dependin kaltainen mittareita generoiva työkalu. PHPLOC on ensisijaisesti projektin kokoa kartoittava työkalu. PHPLOC mittaa projektin koodi- ja kommenttirivien lisäksi luokkien, rajapintojen ja muiden rakenteiden määriä.

PHP Mess Detector on PHP_Depend-sovelluksen pohjalta luotu staattisen analyysin työkalu. PHPMD pyrkii tarjoamaan PHP-projekteille saman toiminnallisuuden kuin PMD-työkalu Javalle. Ohjelmistomittareiden generoimisen sijaan PHPMD etsii konkreettisia ongelmia lähdekoodista. Ongelmat voivat olla esimerkiksi ohjelmointivirheitä, ongelmia luokkahierarkiassa tai käyttämättömiä parametreja tai metodeja. PHPMD havaitsee lisäksi liian monimutkaiset ilmaisut ja varoittaa huonosti nimetyistä muuttujista. Työkalu toimii sääntöjen mukaan, jotka määritellään XML-tiedostoissa. Valvottavat säännöt valitaan sääntökokoelmista. PHPMD:n jakelun mukana tulee valmiita sääntökokoelmia, jotka soveltuvat hyvin useimpiin projekteihin. Sääntöjä voi myös muokata projektin tarpeisiin sopiviksi. PHPMD on hyvä lisä lähes kaikkien PHP-projektien kehitystyöhön, koska tavallisista ohjelmistomittareista ei voi suoraan vetää johtopäätöksiä koodin laadusta. Mittarit vaativat huolellista analysointia, eikä yksittäisen projektin mittarit ole täysin vertailukelpoisia muiden projektien vastaaviin mittareihin. Tästä poiketen PHPMD:n varoitukset ovat hyvin selkeitä ja yksiselitteisiä. Jos varoitukset ovat aiheettomia, täytyy työkalun asetuksiin tehdä muutoksia. Muussa tapauksessa ilmoitetut ongelmat voi välittömästi korjata.

Ohjelmistoprojekteissa halutaan usein noudattaa tiettyä ohjelmointityyliä, joka standardoidaan organisaation tai projektin laajuisesti. Jokaisen projektin kehittäjän täytyy siis noudattaa samaa tyyliä. Organisaatioissa koodityyli on yleensä kaikissa samalla ohjelmointikielellä toteutetuissa projekteissa sama. Tyylimäärittelyihin voi kuulua esimerkiksi rivinvaihtojen paikat, välilyöntien ja sarkainmerkkien käyttö, kommenttien tyyli ja muuttujien nimeämiskäytännöt. Ohjelmointityylit vaihtelevat hyvin paljon kehittäjien välillä. Hyvää ohjelmointityyliä on vaikea määritellä, koska asia on hyvin subjektiivinen. Ohjelmointityylin standardoiminen lisää tuottavuutta, koska koodi on yhtenäistä ja helposti luettavaa. PHP_CodeSniffer on työkalu ohjelmointityylin standardoimiseksi. Työkalulle määritellään halutunlainen tyyli, ja työkalu ilmoittaa koodissa olevista poikkeuksista. Kuten monet muut työkalut, PHP_CodeSniffer on saanut vaikutteita Java-

maailmasta. Javalla toimiva vastaava työkalu on Checkstyle. PHP_CodeSniffer osaa tarvittaessa tuottaa XML-muotoisen raportin samassa formaatissa kuin Checkstyle. Tätä raporttia voi helposti hyödyntää Jenkins CI:n kaltaisella koontipalvelimella.

Huonosti suunnitellussa projektissa saattaa syntyä tilanteita, joissa kehittäjä päätyy kopioimaan suuria koodilohkoja esimerkiksi luokkien välillä. Täydellisten funktioiden tai suurien koodilohkojen kopiointi viittaa usein huonoon suunnitteluun ja projektin rakenteeseen. Kopioinnista syntyy tilanteita, joissa alun perin täsmälleen samaa koodia ylläpidetään kahdessa tai useammassa paikassa. PHPCPD on työkalu kopioidun koodin tunnistamiseen. PHPCPD ilmoittaa kopioinneista, jotka ylittävät kehittäjän määrittelemät kynnysarvot. Kopioidut osiot kannattaa refaktoroida omaksi kokonaisuudekseen erilliseen luokkaan tai funktioon.

6.2 Antin käyttäminen PHP-koontiin

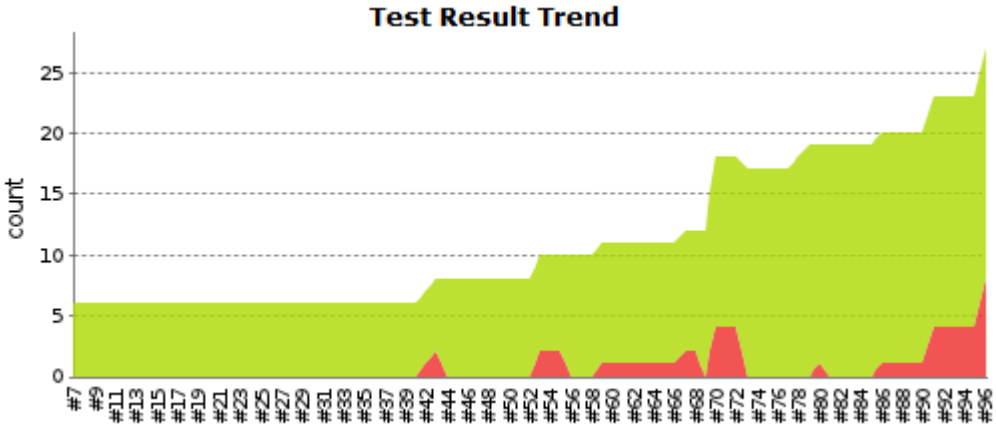
Sebastian Bergmann on kehittänyt Antille valmiin prosessikuvauksen, jonka avulla Ant saadaan suorittamaan kaikki luvussa 6.1 mainitut koontityökalut [Bergmann 2011b]. Koontityökalujen suorittaminen on toteutettu yksinkertaisesti Antin exec-tehtävää käyttämällä. Valmiin prosessikuvauksen ongelmana on sen toimimattomuus Windowsissa. Windowsin komentosarjatiedostoja ei voi kutsua suoraan Antin exec-tehtävästä, vaan kutsu täytyy tehdä Windowsin komentorivitulkkiin ja käyttää /c-vipua komentosarjatiedoston kutsumiseen. Antin exec-tehtävän käyttöä ja edellä mainittua ongelmaa havainnollistetaan koodiesimerkissä 18.

```
<!-- *nix-järjestelmät-->
<target name="phpunit">
    <exec executable="phpunit" failonerror="true"/>
</target>

<!-- Windows -->
<target name="phpunit">
    <exec executable="cmd" failonerror="true">
        <arg line="/c phpunit"/>
    </exec>
</target>
```

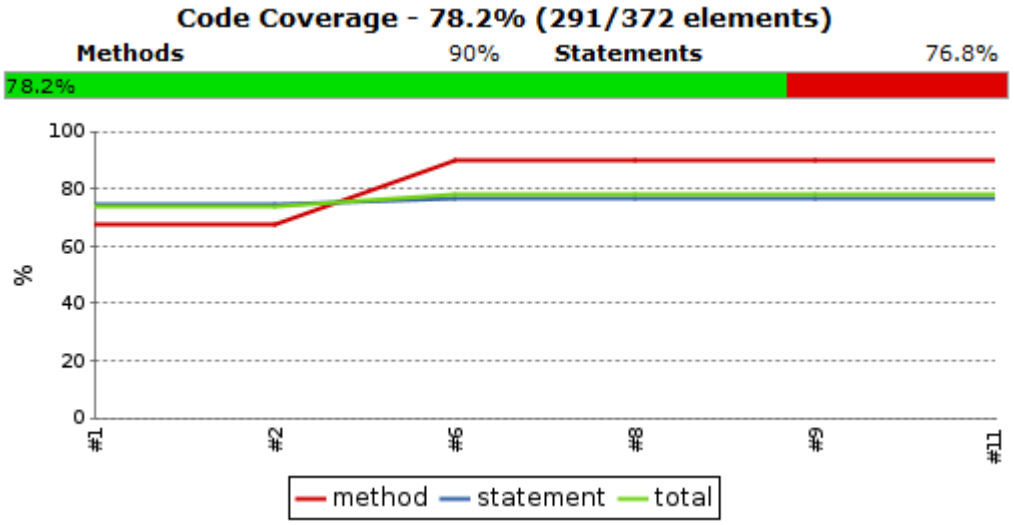
Koodiesimerkki 18. Komentosarjatiedostojen kutsuminen eri käyttöjärjestelmissä Antin exec-tehtävällä.

Ongelman voisi ratkaista muuttamalla prosessikuvausta; Antin exec-tehtävät voi tehdä alustariippumattomaksi ehtorakenteella. Ratkaisu on prosessikuvauksen luettavuudelle haitallinen, koska ehtorakenteiden käyttäminen Antin XML-pohjaisella kielellä on varsin kankeaa. Tyypillinen koontityökaluissa käytetty ratkaisu olisi exec-tehtävien abstrahoiminen liitännäisen taakse. Tätä lähestymistapaa käytetään luvussa 6.4 esiteltävässä Gradle-liitännäisessä. Liitännäistä käyttävä prosessikuvaus näyttäisi kehittäjälle yksinkertaisemmalta ja työkalujen asetukset olisi helpompi määritellä. Toisaalta tämä vaatisi ulkoisen riippuvuuden lisäämisen Ant-prosessikuvaukseen, eikä Antissa ole sisäänrakennettua riippuvuuksienhallintaa.

Valmiin Ant-prosessikuvauksen ainoana tarkoituksena ei ole helpon tien tarjoaminen PHP-laaduntarkastustyökalujen suorittamiseen. Prosessikuvauksen lisäksi Bergmann on tehnyt Jenkinsille valmiin työpohjan, joka hyödyntää prosessikuvauksen työkalujen tuottamia raportteja. XML- ja CSV-muotoiset raportit esitetään Jenkinsissä kuvaajina tai taulukoituna. Kuvissa 10 ja 11 on esitetty PHPUnitin testituloskuvaaja ja testikattavuuskuvaaja.



Kuva 10. Yksikkötestien tuloskuvaaja.



Kuva 11. Testikattavuuskuvaaja.

Valmiin Ant-prosessikuvauksen ja Jenkins-työpohjan ansiosta jatkuvan integroinnin käyttöönotto on PHP-ympäristössä helppoa. Laaduntarkastustyökalujen konfigurointi vaatii jonkin verran perehtymistä, mikäli niiden oletusasetukset eivät sovi projektille. Erityisesti PHP_CodeSniffer vaatii konfigurointia, koska koodin tyyliseikat ovat subjektiivisia ja käytännöt vaihtelevat eri kehittäjäryhmien välillä paljon. Metropolian ohjelmistotuotantoprojekteissa tyyliseikkoihin ei juuri ehditä kuluttaa aikaa, joten PHP_CodeSnifferin hyöty jää niissä kyseenalaiseksi.

Valmis prosessikuvaus on helppo ottaa käyttöön, mutta se ei palvele kaikkia käyttäjiä. Bergmann on kehittänyt PHP Project Wizard -työkalun käyttötapauksiin, joihin valmis prosessikuvaus ei sovi. PPW generoi Ant-prosessikuvauksen, joka kutsuu laaduntarkastustyökaluja projektiin sopivilla parametreilla. PPW:lle syötetään parametreina tietoa projektin hakemistorakenteesta. Muilta osin PPW ei vaikuta generoituun prosessikuvaukseen, joten se on sisällöltään lähes identtinen valmiin prosessikuvauspohjan kanssa. Tämä tarkoittaa sitä, että työkalujen yksittäisiä parametreja joudutaan edelleen muokkaamaan suoraan prosessikuvauksesta käsin. PPW on tästä huolimatta helpoin ja nopein tapa luoda omalle projektille soveltuva prosessikuvaus. Liitteessä 2 on esitetty PPW:n käyttö ja esimerkit sen generoimista tiedostoista.

6.3 Maven for PHP

Maven for PHP on Maven-liitännäinen, jonka kunnianhimoisena tavoitteena on olla kokonaisvaltainen koontityökalu PHP-sovelluksille. Sovellusten koonnin lisäksi liitännäinen pyrkii tarjoamaan riippuvuuksienhallinnan ja projektinhallintaan liittyviä ominaisuuksia. Liitännäisen toteutus on uskollinen Mavenin perusidealle. Koonti suoritetaan Mavenin elämänkaarimallin mukaisesti ja riippuvuudet määritellään samalla periaatteella kuin esimerkiksi Java-projekteissa. Riippuvuuksienhallinnan tavoitteena on tukea yksittäisten kirjastojen asentamisen lisäksi myös kokonaisten sovelluskehysten hallinnoimista. Maven for PHP on mielenkiintoinen projekti, mutta sen nykyiseen toteutukseen liittyy monia ongelmia.

PHP-sovellusten riippuvuuksienhallinnan integroiminen koontityökaluun on erittäin haastava tehtävä. Uudelleenkäytettäviä PHP-komponentteja jaellaan usein phar-pakkauksissa. Phar-pakkausten lisäksi sovellusten ja kirjastojen jakeluun on kehitetty nykyisin suosittu ja laajalti käytetty paketinhallintajärjestelmä, PEAR (PHP Extension and Application Repository). Maven for PHP sisältää alkukantaisen tuen phar- ja PEAR-pakkauksille, mutta niiden käsittelyyn liittyy huomattavia ongelmia. Vaikka PEAR-pakkauksenhallintajärjestelmä on yhtenäinen jakelukanava, ei pakkausten sisältöä ole standardoitu kovin pitkälle. Osa PEAR-pakkauksista on puhtaita PHP-kirjastoja, mutta useat pakkaukset koostuvat komentoriviskripteistä ja dokumentaatiosta. Maven for PHP:n kaltainen työkalu ei voi helposti asentaa PEAR-pakkauksia projektiin, koska niiden käyttöönottoon liittyviä vaiheita ei voi päätellä pakkauksista. Komponentit saattavat vaatia alustustehtäviä, kuten esimerkiksi tietokannan käyttöönoton. Tämänkaltaisten tehtävien automatisoiminen on hyvin vaikeaa, eikä Maven for PHP tarjoa ongelmaan ratkaisua. [Eisengardt 2012.]

Riippuvuuksienhallintaan liittyvät ongelmat pätevät myös Maven for PHP:n sovelluskehyksien tukeen. Yksinkertaisimmillaan PHP-sovelluskehys koostuu ainoastaan luokkakirjastosta. Esimerkiksi suosittu Zend Framework sisältää ainoastaan luokkakirjaston ja näkymäskriptejä. Tällaisen kehyksen käyttöönoton automatisointi onnistuu koontityökalulla helposti. Ongelmallisia ovat sovelluskehykset, jotka sisältävät paljon omia käytäntöjä ja ulkoisia hallintatyökaluja. Monet sovelluskehykset hyödyntävät ulkoista pakkauksenhallintajärjestelmää, joka on vastuussa kehyksen moduulien lataamisesta. PHP-sovelluskehysten käyttämät hakemistorakenteet eivät ole standardoituneet samalla

tavalla kuin Mavenin oletuksena käyttämä rakenne Java-maailmassa. Tavallista on myös komentorivisovelluksen käyttäminen sovelluskehyksellä luodun projektin hallinnoimiseen. Näistä syistä PHP-sovelluskehysten käyttöönotto ja hallinnoiminen koontityökaluilla on hyvin vaikeaa, eikä Maven for PHP:n nykyinen versio pysty ratkaisemaan riippuvuuksienhallintaan liittyviä ongelmia. Liitännäisen kehittäjien tavoitteena on tukea tulevissa versioissa suosituimpia sovelluskehyksiä. Kaikkien sovelluskehyksien tukeminen olisi mahdotonta.

Maven for PHP -liitännäinen pystyy kutsumaan luvussa 6.1 käsiteltyjä laaduntarkastustyökaluja. Maven-prosessikuvaus työkalujen kutsumiseen on hyvin monisanainen, eikä se tarjoa juuri mitään etua Antin vastaavaan prosessikuvaukseen. Maven for PHP on nykymuodossaan lähinnä kuriositeetti, jonka käytännön hyödyt jäävät vähäisiksi. Projektin saattaa kuitenkin tulevaisuudessa nousta varteenotettavaksi vaihtoehdoksi PHP-koontiin, joten sen kehitystä kannattaa seurata.

6.4 Gradle-liitännäinen

Luvussa 3.5 esiteltiin Gradle ja todettiin sen olevan erittäin hyvä yleiskäyttöinen koontityökalu. Kuten Antia, Gradleä voi käyttää koontityökaluna monien ohjelmointikielien ja -ympäristöjen kanssa. Yleiskäyttöisyydestä huolimatta Gradlen kehityksessä keskitytään ensisijaisesti Java-ekosysteemiin. Tämä heijastuu myös liitännäisiin, joista valtaosa liittyy Java-perheen kieliin ja työkaluihin. PHP-koonti onnistuu Phingillä helpommin kuin Gradlellä, koska Phing tukee suoraan monia PHP-koonnissa käytettäviä työkaluja. Phingin prosessikuvauksissa voi melko helposti kuvata tehtäviä, jotka suorittavat kyseisiä työkaluja. Jotta Gradle pystyisi tarjoamaan helposti lähestyttävän tavan PHP-koontiin, tarvitsisi se liitännäisen, joka piilottaisi PHP-laaduntarkastustyökalujen kutsumiseen liittyvät yksityiskohdat. Gradlen joustavan ohjelmointirajapinnan ansiosta tällaisen liitännäisen toteuttaminen ei ole kovin vaikeaa. Tässä luvussa esitetään PHP-koontiin tarkoitetun Gradle-liitännäisen toteutus. Liitännäinen kehitettiin prototyyppinä, jonka tarkoituksena oli kartoittaa Gradlen soveltuvuutta PHP-koontiin. Liitännäiseen ei siis toteutettu tukea läheskään kaikille laaduntarkastustyökaluille.

Liitännäiseen valittiin tuettaviksi työkaluiksi PHPUnit ja PHP Mess Detector. Molemmat työkalut ovat komentorivisovelluksia, joten niiden kutsuminen koontityökaluista on

helppoa. Liitännäisen tehtäväksi jää komentoriviparametrien määritteleminen. Parametrien määritteleminen prosessikuvauksessa täytyy olla intuitiivista ja helppoa. Esimerkiksi Antin exec-tehtävien muokkaaminen on hyvin virhealtista ja vaatii yksittäisten työkalujen dokumentaatioon perehtymistä. Gradle-liitännäisen kehityksen tavoitteena oli Java-koonnin kaltaisen yksinkertaisuuden tuominen PHP-koontiin. Koonnin täytyy siis perustua oletusarvoihin, joita prosessikuvauksessa voi helposti muokata. Liitännäiseen ei ohjelmoitu tukea kaikille PHPUnitin ja PHPMD:n ominaisuuksille. Puuttuvien ominaisuuksien lisääminen onnistuisi suhteellisen pienellä työpanoksella.

Liitännäisen ohjelmakoodi ja projektin rakenne on esitetty liitteessä 1. Huomionarvoista on, että liitännäinen ohjelmoitiin Groovylla. Gradle-liitännäiset voi ohjelmoida millä tahansa Java-tavukoodiksi kääntyvällä kielellä. Groovyn lisäksi hyviä vaihtoehtoja ovat periaatteessa Java ja Scala. Käytännössä Gradlen ohjelmointirajapinta osoittautui hankalakäyttöiseksi Javalla. Java-koodia tarvitaan Groovyyn verrattuna moninkertainen määrä saman toiminnallisuuden toteuttamiseksi. Rajapinta on rakennettu siten, että Groovyn sulkeumia ja muita funktionaalisen ohjelmoinnin ominaisuuksia käyttämällä liitännäisiä pystytään ohjelmoimaan tiiviillä ja helposti luettavalla koodilla. Vastaavasti Java-koodista tulee erittäin monisanaista ja vaikealukuista.

Liitännäisen toteutus onnistui hyvin. Sitä käyttämällä Gradlellä pystytään kutsumaan PHPUnitia ja PHP Mess Detectoria yksinkertaisella prosessikuvauksella. Liitteen 1 koodiesimerkeissä 4 ja 6 kuvatut luokat asettavat PHPUnitin ja PHPMD:n kutsujen oletusparametrit. Oletusarvot helpottavat liitännäisen loppukäyttäjän työtä prosessikuvauksen laatimisessa. Koodiesimerkissä 19 on esitetty yksinkertainen prosessikuvaus, jossa liitännäistä hyödynnetään.

```
buildscript {
    repositories {
        maven {
            url "http://artifactory.metropolia.fi/repo"
        }
    }

    dependencies {
        classpath group: 'fi.metropolia', name: 'phpqa', version: '0.1.0-SNAPSHOT'
    }
}

apply plugin: 'phpqa'
```

Koodiesimerkki 19. Yksinkertainen prosessikuvaus, joka käyttää laaduntarkastusliitännäistä.

Esimerkin buildscript-lohkossa määritellään riippuvuus liitännäisestä. Prosessikuvauksen viimeinen rivi ottaa liitännäisen varsinaisesti käyttöön. Mitään muita määrittelyjä ei tarvita, mikäli projektin hakemistorakenne noudattaa liitännäiseen ohjelmoituja oletusarvoja, ja PHPUnitia ja PHP Mess Detectoria halutaan kutsua niin ikään liitännäisen määrittelemillä parametreilla.

Liitännäisten oletusasetukset ovat käytännöllisiä, kun ne palvelevat suurta osaa käyttäjistä. Tämän lisäksi niiden täytyy myös olla helposti muutettavissa. Laaduntarkastusliitännäinen onnistuu tässä hyvin. Sen asetuksia voi kätevästi muokata Gradlen sovellusaluekohtaisella kielellä. Koodiesimerkissä 20 havainnollistetaan liitännäisen konfiguroimista.

```
buildscript {
    repositories {
        maven {
            url "http://artifactory.metropolia.fi/repo"
        }
    }

    dependencies {
        classpath group: 'fi.metropolia', name: 'phpqa', version: '0.1.0-SNAPSHOT'
    }
}

apply plugin: 'phpqa'

phpmd {
    sourceDir = "application"
    reportFormat = "html"
    ruleSet = "codesize,naming"
}

phpunit {
    cloverReport = false
    junitReport = "${project.buildDir}/junit.xml"
    bootstrap = "tests/bootstrap.php"
    haltOnError = true
    haltOnFailure = false
    processIsolation = false
    strict = false
    verbose = true
}
```

Koodiesimerkki 20. Laaduntarkastusliitännäistä hyödyntävä prosessikuvaus, jossa poiketaan liitännäisen oletusarvoista.

Osa esimerkin asetuksista on samoja kuin oletusarvot, mutta tästä ei ole haittaa. Kaikkia asetuksia ei ole myöskään eksplisiittisesti määritelty, jolloin ne ovat edelleen oletusarvoissaan.

6.5 Yhteenveto PHP-koonnista

PHP-koontiin erikoistuneita työkaluja ei Phingin lisäksi juuri ole. Phing on hyvä koontityökalu pienelle kohderyhmälle, mutta useimmille käyttäjille Ant on tutumpi työkalu, joka tarjoaa lähes samat ominaisuudet. Sebastian Bergmannin työkalujen ansiosta Ant on helpommin lähestyttävä työkalu PHP-koontiin, kun hyödynnetään jatkuvaa integrointia. Antin PHP-koonnin käyttöönotto Jenkinsin kanssa on vaivatonta, mikä onkin tärkeää, sillä jatkuvan integroinnin käyttämisen suurin kynnys on sen käyttöönottoon tarvittava suurehko työmäärä. Ant on tällä hetkellä kokonaisuutena paras työkalu PHP-koontiin jatkuvan integroinnin kanssa, mutta tilanne saattaa muuttua lähitulevaisuudessa. Sekä Ant että Phing kärsivät XML-muotoisten prosessikuvausten ongelmista, eivätkä ne sisällä ominaisuuksia, joiden ansiosta Maven ja Gradle ovat Java-koonnissa suuressa suosiossa.

Tulevaisuudessa Antin sijaan PHP-koontiin saatetaan käyttää Gradleä tai Mavenia. Maven for PHP -liitännäinen on lupaava projekti, joka saattaa mullistaa PHP-koonnin. Muut koontityökalut eivät tarjoa minkäänlaista riippuvuuksienhallintaa PHP-projekteille. Toisaalta monet muutkin modernit koontityökalut ovat hyviä alustoja liitännäiskehitykselle. Gradlelle toteutettu yksinkertainen liitännäinen osoitti, että Gradle on varteenotettava vaihtoehto lähes mihin tahansa koontitehtävään. Liitännäistarjonta ei vielä vastaa Mavenia tai Antia, mutta Gradle on noussut nopeasti kolmanneksi suosituimmaksi koontityökaluksi Java-maailmassa.

7 Yhteenveto

Tässä opinnäytetyössä tutkittiin jatkuvan integroinnin koontityökaluja ja tietovarastonhallintasovelluksia. Tarkoituksena oli tarkastella uusien, suosiota saavuttaneiden, työkalujen soveltuvuutta Metropolia Ammattikorkeakoulun ohjelmistotuotantoprojekteihin. Koontityökaluja tarkasteltiin yleisluontoisesti ja tarkemmiksi tutkimuskohteiksi valittiin Android- ja PHP-sovellusten koonti. Metropolian ohjelmistoprojekteissa on kehitetty sovelluksia molemmille alustoille, mutta jatkuvan integroinnin kehitysympäristö rakennettiin alun perin tukemaan Java-koontia, joka suoritetaan Antilla tai Mavenilla. Jatkuvaa integrointia ei siis ole kunnolla pystytty hyödyntämään Android- ja PHP-projektien kanssa. Tämän työn tarkoituksena oli kartoittaa uusien koontityökalujen soveltuvuutta

kehitysympäristöön ja tutkia, miten PHP- ja Android-sovelluskehitys saadaan parhaiten liitettyä ympäristöön. Lisäksi tarkasteltiin tietovarastonhallintasovelluksia, jotka ovat olennaisia pitkälle viedyssä riippuvuuksienhallinnassa.

Uusia koontityökaluja päädyttiin tarkastelemaan Antiin ja Maveniin kohdistuneen kritiikin vuoksi. Antin imperatiiviset prosessikuvaukset ovat hyvin joustavia, mutta myös monisanaisia ja vaikeita ylläpitää. Mavenin jäykät käytännöt hankaloittavat koontilogiikan ohjelmointia ja tekevät Mavenin käyttämisestä vaikeaa. Monet ongelmat ovat lähtöisin XML-pohjaisista sovellusaluekohtaisista kielistä, joita Ant ja Maven käyttävät. Tässä työssä esitetyt prosessikuvaukset havainnollistavat tehokkaasti XML-pohjaisten sovellusaluekohtaisten kielien ongelmia. Syntaksikorostuksesta huolimatta Antin ja Mavenin prosessikuvaukset ovat hankalalukuisia ja monisanaisia. Tällä saralla vaihtoehtoisten koontityökalujen tarkastelu osoittautui hedelmälliseksi. Lähes kaikki tutkitut koontityökalut käyttävät XML-pohjaisen kielen sijaan ohjelmointikieleen perustuvaa sovellusaluekohtaista kieltä. Ratkaisu on selkeästi oikea, eikä XML:ää tulla näkemään tulevaisuuden koontityökalujen konfiguroinnissa.

Monet modernit koontityökalut onnistuvat XML:ään liittyvien ongelmien ratkaisemisen lisäksi yhdistämään Antin ja Mavenin parhaat puolet yhdeksi toimivaksi kokonaisuudeksi. Lupaavimpia työkaluja ovat Buildr ja Gradle. Molempien ohjelmointikieliin perustuvat sovellusaluekohtaiset kielet tekevät prosessikuvauksista helppolukuisia ja tiiviitä, mutta myös erittäin tehokkaita. Erityisesti Gradle osoittautui hyvin kypsäksi koontityökaluksi, joka pystyy periaatteessa korvaamaan Antin ja Mavenin täysin. Gradle sisältää lisäksi omia innovaatioita, joita muilla koontityökaluilla ei ole tarjota. Android- ja PHP-koontiin kohdistuneet tutkimustapaukset kuitenkin osoittivat, että sekä Antilla että Mavenilla on vielä paikkansa jatkuvan integroinnin koontityökaluina. Antin osalta molemmissa tutkimustapauksissa ratkaisevaksi tekijäksi osoittautuivat aputyökalut, jotka helpottavat sen käyttöönottoa merkittävästi. Myös Maven tarjoaa joitain etuja muihin koontityökaluihin verrattuna. Se integroituu koontityökaluista parhaiten kehitysympäristöihin ja sille on liitännäisiä lähes mihin tahansa tarpeeseen. Metropolian ohjelmistotuotantoprojekteissa tullaan hyödyntämään tulevaisuudessakin Antia ja Mavenia. Muita työkaluja, kuten Gradleä, saatetaan käyttää projekteissa, joissa Ant tai Maven eivät tarjoa merkittäviä etuja.

Koontityökalujen tulevaisuuden näkymät ovat mielenkiintoisia. Ant ja Maven eivät välttämättä tule olemaan suosituimpia koontityökaluja pitkään Java-maailmassa, sillä modernien työkalujen edut ovat melko selviä. Uusien koontityökalujen suosion kasvu on kuitenkin melko hidasta. Osasyynä saattaa olla ylitarjonta samoja ominaisuuksia tarjoavista työkaluista. Buildr ei tarjoa Gradleen verrattuna juuri mitään etua, mutta molemmilla työkaluilla on silti omat käyttäjäkuntanasa. Mavenin kaltaiset menestystarinat pohjautuvat suureen käyttäjäkuntaan, joka kehittää liitännäisiä ja apuohjelmia koontityökalulle. Uudet koontityökalut ovat tuntemattomia, eikä niille ole saatavilla aktiivisesti kehitettäviä liitännäisiä kaikkiin tarpeisiin.

PHP-koontiin tehty tutkimus osoitti, että PHP-projektien jatkuva integrointi on vielä suurelta osin ratkaisematon ongelma. Antilla toteutettu koonti on välttävä ratkaisu, joka on Java-maailman mahdollisuuksiin verrattuna hyvin alkeellinen. Phing ei ratkaise Antin perusongelmia, joten sen hyödyllisyys rajoittuu harvoihin tapauksiin. Maven for PHP -liitännäinen saattaa tulevaisuudessa ratkaista PHP-projektien riippuvuuksienhallinnan ongelman. Se on myös ainoa projekti, joka pyrkii tuomaan deklaratiivisen prosessikuvaustyylin PHP-koontiin. Liitännäisen nykyiseen toteutukseen liittyvät ongelmat tekevät siitä huonon valinnan koontityökaluksi, mutta projekti on seuraamisen arvoinen. Gradlelle tehty PHP-laaduntarkastusliitännäinen osoittaa, että modernit koontityökalut pystyisivät sopivilla liitännäisillä kilpailemaan helposti Antin ja Mavenin kanssa.

Metropolian ohjelmistotuotantoprojekteissa käytetään paljon Java-perheen kieliä ja sovelluskehyksiä. Näissä projekteissa hyödynnetään lähes poikkeuksetta riippuvuuksienhallintaa. Monissa tapauksissa riippuvuuksienhallinnasta ei saada täyttä hyötyä ilman organisaation sisäistä tietovarastopalvelinta. Tietovarastonhallintasovellukset sopivat hyvin jatkuvaan integrointiin ja Artifactory toimii saumattomasti koontipalvelimien kanssa. Tietovarastopalvelinta ei liitetty tämän työn tekemisen aikana Metropolian kehitysympäristöön, mutta se saattaa tulla kyseeseen tulevaisuudessa.

Lähteet

ADT Plugin for Eclipse. Verkkosivu. <http://developer.android.com/sdk/eclipse-adt.html>. Luettu 7.4.2012.

Antwrap. 2008. Verkkosivu. <https://github.com/atoulme/Antwrap/blob/master/README.txt>. Päivitetty 17.2.2008. Luettu 15.12.2011.

Ant: A Critical Retrospective. 2011. Verkkosivu. <http://ebuild-project.org/articles/a_critical_retrospective_ant.html>. Päivitetty 24.1.2011. Luettu 15.2.2012.

Apache Ant - Frequently Asked Questions. 2011. Verkkosivu. <http://ant.apache.org/faq.html#history>. Päivitetty 7.8.2011. Luettu 30.11.2011.

Barnette, John. 2009. On Rake. Verkkosivu. <http://www.jbarnette.com/2009/08/27/on-rake.html>. Päivitetty 27.8.2009. Luettu 21.11.2011.

Bayer, Andrew. 2011. Hudson's future. Verkkosivu. <http://jenkins-ci.org/content/hudsons-future>. Päivitetty 11.1.2011. Luettu 24.11.2011.

Ben Haim, Shlomi. 2011. Artifactory Vs. Nexus The Integration Matrix. Verkkosivu. <http://blogs.jfrog.org/2011/02/artifactory-vs-nexus-integration-matrix.html>. Päivitetty 24.2.2011. Luettu 4.1.2012.

Berglund, Tim & McCullough, Matthew. 2011. Building and Testing with Gradle. Sebastopol, California: O'Reilly Media. E-kirja.

Bergmann, Sebastian. 2006. PHPUnit Pocket Guide. Sebastopol, California: O'Reilly Media.

Bergmann, Sebastian. 2011a. Integrating PHP Projects with Jenkins. Sebastopol, California: O'Reilly Media. E-kirja.

Bergmann, Sebastian. 2011b. Template for Jenkins Jobs for PHP Projects. Verkkosivu. <http://jenkins-php.org/>. Luettu 8.3.2012.

Building and Running. Verkkosivu. <http://developer.android.com/guide/developing/building/index.html>. Luettu 26.3.2012.

Buildr. Verkkodokumentti. <http://buildr.apache.org/buildr.pdf>. Luettu 16.12.2011.

Buildr::Project. Verkkosivu.
<http://buildr.apache.org/rdoc/classes/Buildr/Project.html>. Luettu 15.12.2011.

Dockter, Hans & Murdoch, Adam. 2012a. Dependency Management. Verkkosivu.
<http://gradle.org/docs/current/userguide/dependency_management.html>. Päivitetty 13.3.2012. Luettu 31.3.2012.

Dockter, Hans & Murdoch, Adam. 2012b. The Gradle Daemon. Verkkosivu.
<http://gradle.org/docs/current/userguide/gradle_daemon.html>. Päivitetty 20.2.2012. Luettu 21.2.2012.

Dockter, Hans & Murdoch, Adam. 2012c. The Gradle Wrapper. Verkkosivu.
<http://gradle.org/docs/current/userguide/gradle_wrapper.html>. Päivitetty 20.2.2012. Luettu 21.2.2012.

Ebersole, Steve. 2010. Gradle: why? Verkkosivu.
<https://community.jboss.org/wiki/GradleWhy>. Päivitetty 5.6.2010. Luettu 19.1.2012.

Eisengardt, Martin. 2012. PHP-Maven 2.1 – Analysis draft. Verkkodokumentti.
<http://www.php-maven.org/php-maven%202.1.pdf>. Luettu 4.4.2012.

Fowler, Martin. 2005. Using the Rake Build Language. Verkkosivu.
<http://martinfowler.com/articles/rake.html>. Päivitetty 10.8.2005. Luettu 21.11.2011.

Fox, Brian. 2009. Contrasting Nexus and Artifactory. Verkkosivu.
<http://www.sonatype.com/people/2009/01/contrasting-nexus-and-artifactory/>. Päivitetty 19.1.2009. Luettu 27.2.2012.

Introduction to the Standard Directory Layout. Verkkosivu. Apache Software Foundation. <http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html>. Luettu 19.1.2012

Java Strategy. 2008. Verkkosivu <http://www.scons.org/wiki/JavaStrategy>. Päivitetty 5.6.2008. Luettu 12.1.2010.

Java Support. 2010. Verkkosivu. <http://www.scons.org/wiki/JavaSupport>. Päivitetty 27.3.2010. Luettu 12.1.2010.

Jenkins Plugins. 2011. Verkkosivu <https://wiki.jenkins-ci.org/display/JENKINS/Plugins>. Päivitetty 15.5.2011. Luettu 17.1.2012.

Landman, Yoav & Simon, Frederic. 2011. Changing the Default Storage. Verkkosivu.
<http://wiki.jfrog.org/confluence/display/RTF/Changing+the+Default+Storage>. Päivitetty 1.5.2011. Luettu 27.2.2012.

Lukkarinen, Aleksi. 2011. Iteratiivinen sovelluskehitys: Kehitysympäristön toteutus ja ylläpito. Opinnäytetyö. Tietotekniikan koulutusohjelma. Metropolia Ammattikorkeakoulu. <http://urn.fi/URN:NBN:fi:amk-201105269796>.

Maven Repository Manager Feature Matrix. 2011. Verkkosivu. <http://docs.codehaus.org/display/MAVENUSER/Maven+Repository+Manager+Feature+Matrix>. Päivitetty 29.12.2011. Luettu 4.1.2012.

Maven: The Complete Reference. Verkkosivu. <http://www.sonatype.com/books/mvnref-book/reference/installation-sect-compare-ant-maven.html>. Luettu 30.11.2011.

Orr, Christopher. 2012. Android Emulator Plugin. Verkkosivu. <https://wiki.jenkins-ci.org/display/JENKINS/Android+Emulator+Plugin>. Päivitetty 27.3.2012. Luettu 10.4.2012.

PMD - Android Rules. Verkkosivu. <http://pmd.sourceforge.net/rules/android.html>. Luettu 10.4.2012.

Repository Management with Maven Repository Managers. Verkkosivu. Apache Software Foundation. <http://maven.apache.org/repository-management.html>. Luettu 21.3.2012.

Repository Management with Nexus. Verkkosivu. Sonatype Inc. <http://www.sonatype.com/books/nexus-book/reference/repoman-sect-reasons.html>. Luettu 29.3.2012.

Sadogursky, Baruch. 2012. Dependency Management with .NET – Doing it Right. Verkkosivu. <http://blogs.jfrog.org/2012/02/dependency-management-with-net-doing-it.html>. Päivitetty 6.2.2012. Luettu 31.3.2012.

Scons Projects. 2011. Verkkosivu. <http://www.scons.org/wiki/SconsProjects>. Päivitetty 23.6.2011. Luettu 12.1.2012.

Smart, John. 2011. Jenkins and Hudson: Butler Wars! Verkkosivu. <http://weblogs.java.net/blog/johnsmart/archive/2011/02/16/jenkins-and-hudson-butler-wars>. Päivitetty 16.2.2011. Luettu 24.11.2011.

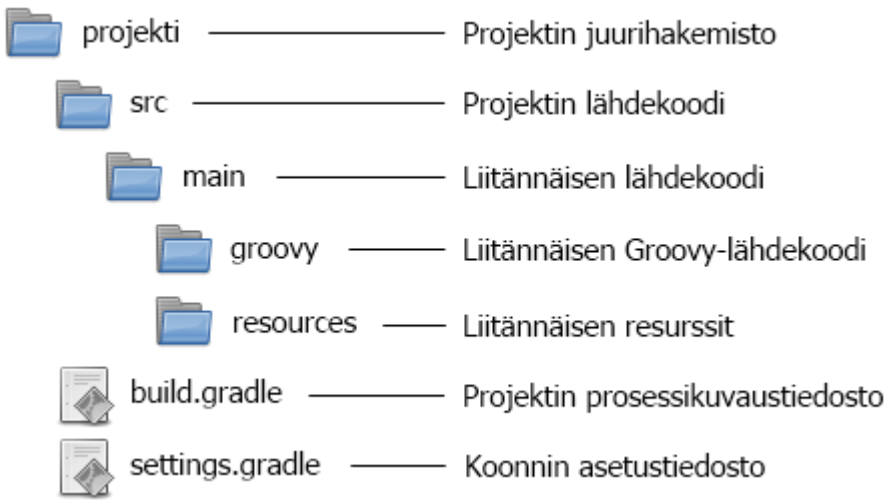
Van Zyl, Jason. 2011. Hudson's Bright Future. Verkkosivu. <http://www.sonatype.com/people/2011/02/hudsons-bright-future/>. Päivitetty 3.2.2011. Luettu 24.11.2011.

Winder, Russel. 2008. Kommentti artikkeliin Ant or Gant? Verkkosivu. <http://java.dzone.com/articles/ant-or-gant-part-1>. Päivitetty 27.9.2008. Luettu 14.1.2012.

Xu, Juven. 2010. How to Back Up Nexus Configuration and Repository Artifacts. Verkkosivu. <http://www.sonatype.com/people/2010/01/how-to-backup-nexus-configuration-and-repository-artifacts/>. Päivitetty 25.1.2010. Luettu 1.4.2012.

Gradle-liitännäinen PHP-projektin laaduntarkastukseen

Tässä liitteessä esitetään Gradlelle toteutetun PHP-laaduntarkastusliitännäisen toteutus. Esitettynä on kaikki projektiin liittyvä ohjelmakoodi ja projektin rakenne. Liitännäinen ohjelmoitiin Groovylla. Projektin hakemistorakenne on esitetty kuvassa 1.



Kuva 1. Liitännäisprojektin rakenne.

Koodiesimerkki 1 on liitännäisen kehityksessä käytetty prosessikuvaus.

```
apply plugin: 'groovy'
apply plugin: 'eclipse'
apply plugin: 'maven'

version = '0.1.0-SNAPSHOT'

description = 'Laaduntarkastusliitännäinen PHP:lle'
group = 'fi.metropolia'

dependencies {
    compile gradleApi()
    groovy localGroovy()
}

eclipse {
    classpath {
        defaultOutputDir = file('build-eclipse')
    }
}
```

Koodiesimerkki 1. Liitännäisen kokoava prosessikuvaus, build.gradle.

Liitännäisen kehitykseen tarvittava prosessikuvaus on hyvin yksinkertainen. Prosessikuvauksen mielenkiintoisinta antia on dependencies-lohko, jossa määritellään projektin riippuvuudet. Esimerkissä riippuvuuksiksi asetetaan Gradlen ohjelmointirajapinta ja paikallinen Groovy-asennus, joka tulee Gradlen jakelun mukana.

Gradle asettaa projektin nimeksi oletusarvoisesti projektin juurihakemiston nimen. Projektin nimi voidaan määritellä erillisessä settings.gradle-tiedostossa, joka sijaitsee projektin juurihakemistossa. Koodiesimerkki 2 havainnollistaa asetuksen määrittelyä.

```
rootProject.name = 'phpqa'
```

Koodiesimerkki 2. Koonnin asetustiedosto, settings.gradle.

Gradlen liitännäisten pääluokan täytyy toteuttaa Plugin-rajapinta. Tämän voi tehdä toteuttamalla Plugin-rajapinnan suoraan tai perimällä jonkin valmiista liitännäisluokista. Tässä toteutuksessa käytettiin BasePlugin-luokkaa, joka lisää projektiin muutamia yleishyödyllisiä tehtäviä. Pääluokka on esitetty koodiesimerkissä 3.

```
class PhpQaPlugin extends BasePlugin {

    static final String PHPQA_GROUP = "PHP laaduntarkastus"
    static final String PHPUNIT_TASK = "phpunit"
    static final String PHPMD_TASK = "phpmd"

    void apply(Project project) {
        super.apply(project)

        project.extensions.phpunit = new PhpUnitExtension(project.buildDir)
        project.extensions.phpmd = new PhpMdExtension(project.buildDir)

        project.task(PHPUNIT_TASK, group: PHPQA_GROUP,
                description: "Suorittaa PHPUnit-testit", type: PhpUnitTask)
        project.task(PHPMD_TASK, group: PHPQA_GROUP,
                description: "Suorittaa PHP Mess Detectorin", type: PhpMdTask)
    }

    static def getOsBaseArguments() {
        if (System.properties['os.name'].toLowerCase().contains('windows')) {
            return ["cmd", "/c"]
        }
        return []
    }

}
```

Koodiesimerkki 3. Liitännäisen pääluokka PhpQaPlugin, PhpQaPlugin.groovy.

Liitännäisillä on usein asetuksia, joita kehittäjä voi muuttaa. Gradlen ohjelmointirajapinnassa paras tapa asetuksien liittämiseen on laajennusten *(extension)* käyttäminen. Laajennukset ovat yksinkertaisia JavaBean-komponentteja. JavaBeanit on helppo toteuttaa Groovylla, koska JavaBeanien vaatimat get- ja set-metodit ovat Groovy-luokissa valmiina. Laajennusluokkien asetuksille voi määritellä oletusarvot, mutta se ei ole pakollista. PHPUnitin ja PHP Mess Detectorin laajennusluokissa oletusarvot on määritelty. Tämä helpottaa liitännäisen käyttöönottoa. Laajennusluokat on esitetty koodiesimerkeissä 4 ja 6.

```
class PhpUnitExtension {

    private File buildDir

    public PhpUnitExtension(File buildDir) {
        this.buildDir = buildDir
    }

    private boolean haltOnError = true
    private boolean haltOnFailure = true
    private boolean processIsolation = false
    private boolean strict = false
    private boolean verbose = false

    private String testsDir = "tests"
    private String bootstrap = testsDir + File.separator + "bootstrap.php"

    private def junitReport = buildDir.path + File.separator + "junit.xml"
    private def cloverReport = buildDir.path + File.separator +
      "clover-coverage.xml"

}
```

Koodiesimerkki 4. PHPUnitin laajennusluokka, PhpUnitExtension.groovy.

Tehtävät ovat keskeisin konsepti Gradlessä. Liitännäiset tuovat projekteihin tehtäviä, joita kehittäjä voi käyttää. PHP-laaduntarkastusliitännäisessä tehtävät sisältävät varsinaisen sovelluslogiikan, joka suorittaa työkalut. Työkaluja kutsutaan parametreilla, jotka on määritetty laajennusluokissa. Tehtäväluokat on esitetty koodiesimerkeissä 5 ja 7.

```
class PhpUnitTask extends DefaultTask {

    def arguments

    @TaskAction
    def runTests() {
        def extension = project.extensions.phpunit
        arguments = PhpQaPlugin.getOsBaseArguments()
        arguments.add("phpunit")

        booleanArgs(extension)
```

```
        bootStrapArg(extension.bootstrap)
        junitReportArg(extension.junitReport)
        cloverReportArg(extension.cloverReport)
        arguments.add(extension.testsDir)

        project.exec { commandLine = arguments }
    }

    def booleanArgs(PhpUnitExtension extension) {
        [haltOnError:"--stop-on-error",
         haltOnFailure:"--stop-on-failure",
         processIsolation:"--process-isolation",
         strict:"--strict",
         verbose:"--verbose"
        ].each() { option, argument ->
                    if (extension.getAt(option)) arguments.add(argument)
                }
    }

    def bootStrapArg(String bootStrapPath) {
        def bootstrapFile = new File(bootStrapPath)

        if (bootstrapFile?.isFile()) {
            arguments.addAll(["--bootstrap", bootstrapFile.path])
        }
    }

    def junitReportArg(junitReportPath) {
        if (junitReportPath) {
            arguments.addAll(["--log-junit", junitReportPath])
        }
    }

    def cloverReportArg(cloverReportPath) {
        if (cloverReportPath) {
            arguments.addAll(["--coverage-clover", cloverReportPath])
        }
    }

}
```

Koodiesimerkki 5. PHPUnit-tehtävän määrittelevä luokka, PhpUnitTask.groovy.

```
class PhpMdExtension {

    private File buildDir

    public PhpMdExtension(File buildDir) {
        this.buildDir = buildDir
    }

    private String sourceDir = "src"
    private String reportFormat = "xml"
    private String reportFile = buildDir.path + File.separator + "pmd.xml"
    private String ruleSet = "codesize,design,naming,unusedcode"
}
```

Koodiesimerkki 6. PHP Mess Detectorin laajennusluokka, PhpMdExtension.groovy.

```
class PhpMdTask extends DefaultTask {
```

```
    @TaskAction
    def runPhpMd() {
        def extension = project.extensions.phpmd
        def arguments = PhpQaPlugin.getOsBaseArguments()

        arguments.addAll("phpmd", extension.sourceDir, extension.reportFormat,
                extension.ruleSet, "--reportfile", extension.reportFile)

        if (!project.buildDir.exists()) {
            project.buildDir.mkdirs();
        }

        project.exec {
            commandLine = arguments
            ignoreExitValue = true
        }
    }

}
```

Koodiesimerkki 7. PHP Mess Detector -tehtävän määrittelevä luokka, PhpMdTask.groovy

Kun liitännäistä käytetään prosessikuvauksessa, täytyy Gradlen löytää liitännäisen pääluokka. Pääluokan nimeä ei voi päätellä liitännäisen nimestä, joten se täytyy erikseen määritellä asetustiedostossa, joka sijaitsee META-INF-hakemiston alla. Tämän liitännäisen tapauksessa täydellinen polku tiedostoon on src/main/resources/META-INF/gradle-plugins/phpqa.properties. Tiedoston sisältöä kuvataan koodiesimerkissä 8.

```
implementation-class=fi.metropolia.phpqa.PhpQaPlugin
```

Koodiesimerkki 8. Asetustiedosto, jonka avulla Gradle löytää liitännäisen, phpqa.properties

PHP Project Wizard -työkalun käyttö

Tässä liitteessä esitetään PHP Project Wizardin käyttö ja sen generoima Ant-prosessikuvaus. Ant-prosessikuvaus luodaan koodiesimerkin 1 komennolla.

```
ppw --name Esimerkki --source application --tests tests --bootstrap
tests/bootstrap.php .
```

Koodiesimerkki 1. Prosessikuvauksen generoiva komento.

Työkalulle annetaan parametreina projektin nimi, lähdekoodihakemisto, testihakemisto ja bootstrap-tiedosto, joka alustaa sovelluksen testejä varten. Viimeisenä parametrina annetaan projektin hakemisto. Esimerkissä projektihakemistoksi on määritelty hakemisto, josta komento suoritetaan. Komento luo projektihakemistoon Ant-prosessikuvauksen ja PHPUnitin konfiguraatiotiedoston, jotka on esitetty koodiesimerkeissä 2 ja 3.

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="Esimerkki" default="build" basedir=".">
 <property name="source" value="application"/>

 <target name="clean" description="Clean up and create artifact directories">
  <delete dir="${basedir}/build/api"/>
  <delete dir="${basedir}/build/code-browser"/>
  <delete dir="${basedir}/build/coverage"/>
  <delete dir="${basedir}/build/logs"/>
  <delete dir="${basedir}/build/pdepend"/>

  <mkdir dir="${basedir}/build/api"/>
  <mkdir dir="${basedir}/build/code-browser"/>
  <mkdir dir="${basedir}/build/coverage"/>
  <mkdir dir="${basedir}/build/logs"/>
  <mkdir dir="${basedir}/build/pdepend"/>
 </target>

 <target name="phpunit" description="Run unit tests using PHPUnit">
  <exec executable="phpunit" failonerror="true"/>
 </target>

 <target name="parallelTasks" description="Run the pdepend, phpmd, phpcpd, phpcs, php
doc and phploc tasks in parallel using a maximum of 2 threads.">
  <parallel threadCount="2">
   <sequential>
    <antcall target="pdepend"/>
    <antcall target="phpmd"/>
   </sequential>
   <antcall target="phpcpd"/>
   <antcall target="phpcs"/>
   <antcall target="phpdoc"/>
```

```
    <antcall target="phploc"/>
   </parallel>
  </target>

  <target name="pdepend" description="Generate jdepend.xml and software metrics charts
  using PHP_Depend">
   <exec executable="pdepend">
    <arg line="--jdepend-xml=${basedir}/build/logs/jdepend.xml
               --jdepend-chart=${basedir}/build/pdepend/dependencies.svg
               --overview-pyramid=${basedir}/build/pdepend/overview-pyramid.svg
               ${source}" />
   </exec>
  </target>

  <target name="phpmd" description="Generate pmd.xml using PHPMD">
   <exec executable="phpmd">
    <arg line="${source}
               xml
               codesize,design,naming,unusedcode
               --reportfile ${basedir}/build/logs/pmd.xml" />
   </exec>
  </target>

  <target name="phpcpd" description="Generate pmd-cpd.xml using PHPCPD">
   <exec executable="phpcpd">
    <arg line="--log-pmd ${basedir}/build/logs/pmd-cpd.xml ${source}" />
   </exec>
  </target>

  <target name="phploc" description="Generate phploc.csv">
   <exec executable="phploc">
    <arg line="--log-csv ${basedir}/build/logs/phploc.csv ${source}" />
   </exec>
  </target>

  <target name="phpcs" description="Generate checkstyle.xml using PHP_CodeSniffer">
   <exec executable="phpcs" output="/dev/null">
    <arg line="--report=checkstyle
               --report-file=${basedir}/build/logs/checkstyle.xml
               --standard=PEAR
               ${source}" />
   </exec>
  </target>

  <target name="phpdoc" description="Generate API documentation using PHPDocumentor">

   <exec executable="phpdoc">
    <arg line="-d ${source} -t ${basedir}/build/api" />
   </exec>
  </target>

  <target name="phpcb" description="Aggregate tool output with PHP_CodeBrowser">
   <exec executable="phpcb">
    <arg line="--log    ${basedir}/build/logs
               --source ${source}
               --output ${basedir}/build/code-browser" />
   </exec>
  </target>

  <target name="build" depends="clean,parallelTasks,phpunit,phpcb"/>
</project>
```

Koodiesimerkki 2. PPW:llä luotu prosessikuvaus, build.xml.

```
<?xml version="1.0" encoding="UTF-8"?>

<phpunit bootstrap="tests/bootstrap.php"
         backupGlobals="false"
         backupStaticAttributes="false"
         strict="true"
         verbose="true">
  <testsuites>
    <testsuite name="Esimerkki">
      <directory suffix="Test.php">tests</directory>
    </testsuite>
  </testsuites>

  <logging>
    <log type="coverage-html" target="build/coverage" title="Esimerkki"
         charset="UTF-8" yui="true" highlight="true"
         lowUpperBound="35" highLowerBound="70"/>
    <log type="coverage-clover" target="build/logs/clover.xml"/>
    <log type="junit" target="build/logs/junit.xml" logIncompleteSkipped="false"/>
  </logging>

  <filter>
    <whitelist addUncoveredFilesFromWhitelist="true">
      <directory suffix=".php">application</directory>
    </whitelist>
  </filter>
</phpunit>
```

Koodiesimerkki 3. PPW:llä luotu PHPUnit-konfiguraatiotiedosto, phpunit.xml.dist.