OULUN SEUDUN AMMATTIKORKEAKOULU

Mikko Väänänen

DEVELOPMENT OF CONTINUOUS INTEGRATION FRAMEWORK FOR EXTERNAL PARTNERS

DEVELOPMENT OF CONTINUOUS INTEGRATION FRAMEWORK FOR EXTERNAL PARTNERS

Mikko Väänänen

Thesis

Spring 2012

Degree Programme in Business Information Systems
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Business Information Systems

Author(s): Mikko Väänänen
Title of thesis: Development of Continuous Integration Framework for External Partners
Supervisor(s): Ilkka Mikkonen
Term and year when the thesis was submitted: Spring 2012
Number of pages: 48

The commissioner of this thesis has ongoing project to deploy continuous integration into their software development. They have outsourced some of their software development to external software suppliers and thus there was a need to bind also the external development into their internal continuous integration system. Their continuous integration services have been built upon open source software, Jenkins that supports extensions in form of plug-ins. Jenkins is based on Java platform.

The commissioner needed a Jenkins plug-in to automate process of downloading software supplier's code from FTP server, building it against the latest code base, committing the source code into commissioner's software repository and uploading the build artifacts back to FTP server. They also needed automatic notifications about the build events via email. Due to software licensing issues the plug-in needed to be implemented as black box where the external software supplier does not need or does not get access to commissioner's source code.

At first technical background information from Jenkins and Java platform were studied, both from professional literature and internet resources from well-known publishers and experts of software development and integration. During year 2011 the requirements of black box plug-in were gathered, the plug-in designed, implemented, tested and finally deployed into production use. During the second half of 2011 more features were implemented and bugs removed.

After deployment in fall 2011 the plug-in became a critical part of software delivery chain of commissioner and has been in production ever since. The feedback from external software supplier has been also very good.

Keywords: software integration, continuous integration, Java, programming

TIIVISTELMÄ

Oulun seudun ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma

Tekijä(t): Mikko Väänänen
Opinnäytetyön nimi: Development of Continuous Integration Framework for External Partners
Työn ohjaaja(t): Ilkka Mikkonen
Työn valmistumislukukausi ja -vuosi: Kevät 2012
Sivumäärä: *48*

Työn toimeksiantajalla on ohjelmistokehityksessään meneillään oleva projekti jatkuvan integroinnin käyttöönotosta. He ovat ulkoistaneet osan ohjelmistokehityksestään ulkoisille ohjelmistotoimittajille, joiden ohjelmistokehitys oli myös tarpeen sisällyttää mukaan toimeksiantajan sisäiseen jatkuvan integroinnin järjestelmään. Toimeksiantajan jatkuvan integroinnin palvelut ovat toteutettu avoimen lähdekoodin ohjelmiston, Jenkinsin, päälle. Jenkins perustuu Java-alustaan ja tukee liitännäisteknologiaa.

Toimeksiantaja tarvitsi Jenkins-liitännäisen automatisoimaan prosessin, jossa ladataan FTP-palvelimelta ohjelmistotoimittajan lähdekoodi, käännetään se yhdessä toimeksiantajan uusimman lähdekoodin kanssa, julkaistaan lähdekoodi toimeksiantajan versionhallintajärjestelmässä ja ladataan käännöksen tuotokset FTP-palvelimelle. Liitännäisen oli myös kyettävä lähettämään automaattisesti sähköpostia käännöstapahtumista. Lisensointisyistä liitännäinen piti toteuttaa mustana laatikkona, jossa ulkoinen ohjelmistotoimittaja ei tarvitse tai ei saa pääsyä toimeksiantajan omistamaan lähdekoodiin.

Aluksi tutkittiin taustatietoa Jenkins-ohjelmistosta sekä Java-alustasta sekä ammattikirjallisuudesta että tunnettujen ohjelmistoalan asiantuntijoiden tuottamista internet-lähteistä. Vuoden 2011 aikana liitännäisen vaatimukset kerättiin yhteen, liitännäinen suuunniteltiin, toteutettiin ja testattiin sekä lopuksi otettiin käyttöön tuotantoympäristössä. Loppuvuodesta 2011 liitännäiseen toteutettiin lisää ominaisuuksia sekä korjattiin vikoja.

Käyttöönoton jälkeen syksyllä 2011 liitännäisestä tuli kriittinen osa toimeksiantajan ohjelmistotoimitusketjua ja se on ollut tuotantokäytössä siitä saakka. Palaute ulkoiselta ohjelmistotoimittajalta on myös ollut todella hyvää.

Asiasanat: ohjelmistointegrointi, jatkuva integrointi, Java, ohjelmointi

CONTENTS

DEFINITIONS AND ABBREVIATIONS 7

1 INTRODUCTION 8

2 SOFTWARE DEVELOPMENT MODELS 10

2.1 Phases of software development 10
2.2 Waterfall model 11
2.3 Agile methods 13
2.4 Test-Driven Development 14

3 CONTINUOUS INTEGRATION 16

3.1 Developer's workflow 16
3.2 Software releasing 18
3.3 Benefits of continuous integration for software project 18

4 JENKINS CI SERVER 20

4.1 About Jenkins 20
4.2 Project background 20
4.3 Distribution and installation 21
4.4 Configuration and integration into enterprise systems 21
4.5 User interface 23
4.6 Job types 24
4.7 Extending Jenkins 25
4.7.1 Built-in components 26
4.7.2 Plug-ins 26
4.8 Scalability 27

5 DEVELOPMENT 29

5.1 Requirements 29
5.2 Schedule 30
5.3 Design 31
5.3.1 Plug-in init 34

5.3.2 Configuration validation ... 34
5.3.3 Poll for new SW ... 37
5.3.4 SW download ... 37
5.3.5 SCM: Pull from repository ... 38
5.3.6 SW builds ... 38
5.3.7 SCM: Commit to repository ... 39
5.3.8 Artifact uploads ... 39
5.3.9 Email notification ... 39
5.4 Implementation and testing ... 40
5.4.1 Agile development method ... 40
5.4.2 Development environment ... 41
5.5 Deployment ... 43
6 CONCLUSIONS AND DISCUSSION ... 44
6.1 About the thesis process ... 44
6.2 Future development plans ... 45
6.3 Documentation ... 46
REFERENCES ... 47

DEFINITIONS AND ABBREVIATIONS

API	Application Programming Interface
CI	Continuous Integration
CPU	Central Processing Unit
FTP	File Transfer Protocol
FTPS	File Transfer Protocol with Security
FTPES	File Transfer Protocol with Explicit Security
IDE	Integrated Development Environment
I/O	Input/Output
LDAP	Lightweight Directory Access Protocol
NFS	Network File System
POM	Project Object Model
SCM	Software Configuration Management
SDK	Software Development Kit
TDD	Test-Driven Development
UI	User Interface
VCS	Version Control System
WAR	Web Application Archive
XML	Extensible Markup Language

1 INTRODUCTION

Due to fierce competition in the global software industry, there is an ongoing need to push software products faster to the market. Thus, improving the software development process can give a company a competitive advantage over its competitors, because according to Martin Fowler, recognized world-class expert in the agile methods and continuous integration (CI), it is usually a most time-consuming part of the productization process of software product. (Duvall, Matyas & Glover 2008.)

In the last ten years agile methods and development models have gained popularity in the software business, previously ruled by sequential waterfall model (Waterfall model, date of acquisition 5 April 2011), because agile development can usually respond faster to continuously changing software requirements (Shore & Warden 2008, 6). Also, continuous feedback from customers is very important (Duvall et al. 2008, 10).

In agile development developers usually submit their changes into software repository several times a day to make their changes visible to other developers in the team (Khalaf & Al-Jedaiah 2008, 1975). Thus, the health and quality of code in the repository is very important, because the work of whole team is based on it (Duvall et al. 2008, 41).

The purpose of CI is to ensure required quality and consistency of the code in the development repository. Usually every code increment submitted by a developer is built with the newest code base and automatically checked against predefined criteria, like static code analysis and automated testing.

The CI systems are automated and designed to work without human intervention. There are several different CI systems, like Jenkins and Cruise Control. This thesis work focuses only on Jenkins, because the commissioner already has ongoing project to deploy Jenkins in their product development

process. Jenkins CI server is an open source software project based on Java platform. It provides a basic CI framework and its features can be easily extended by using Jenkins API and plug-ins (Jenkins API documentation, date of acquisition 17-Apr-2012).

In its current state and with specialized in-house plug-ins Jenkins can be already used in the internal development of commissioner, but it can not be fully utilized with their external partners due to network firewall and legal constraints. For example, there are cases where the external partners are not allowed to access commissioner’s source code due to legal constraints; they should only have access to build artifacts and possibly test results. Thus, there is a need to develop a custom plug-in for Jenkins to implement the black box functionality, where the CI server would download the external partner's code from predefined location, build and test it with commissioner’s code base and then upload the build artifacts back to external partner.

The development task of this thesis is to develop a black box plug-in using Jenkins API and its plug-in architecture. The plug-in will be developed with Java language and it will be deployed into commissioner’s development process after the planned functionality has been implemented and tested. There will also be a pilot phase with one external partner. The plug-in will enable external partner to release software to commissioner even several times a day. It will also be easy to monitor the quality of software deliveries through the automated testing, which is presumed to lead to better software quality.

2 SOFTWARE DEVELOPMENT MODELS

Software model can be seen as very important part of organized software product development, because it defines a solid framework and high level description of how the lifecycle of software product is managed.

Following chapters will describe the basic principles of two popular development models: the waterfall model and agile methods, because they are the only relevant ones in context of the commissioner of this thesis work. Also, of all the lifecycle phases of software product, this thesis focuses mostly on integration and releasing.

2.1 Phases of software development

Although there are lot of different software development models and processes, there are certain phases that are common to all of them. They might appear in different order or in different form, but are still employed in professional software project. According to Langr (2005, 10), these phases are:

- Analysis
- Planning
- Design
- Coding
- Testing
- Deployment
- Documentation
- Review

In the analysis phase the project requirements are gathered and refined; what should be built and what the software should do. During planning the project schedule is built and project dependencies sorted out. Design phase focuses on architectural issues, how the software components are organized in a system and how they work together. (Langr 2005, 10.)

Coding, or implementation as it is often called can be started as soon as the design is in place. In test-driven development testing is tightly integrated into the implementation, but e.g. waterfall model has its own well-defined testing phase. One could argue that integration should be also listed in these common phases, but it can be though to be included in the implementation phase. (Langr 2005, 10.)

When the software is deemed ready, it is deployed into use. At this phase the documentation should be also quite ready. Finally, there is usually some kind of review phase where the feedback is gathered and project closed. (Langr 2005, 10.)

2.2 Waterfall model

The classic waterfall model was the only widely accepted software development model until the early 1980s (Waterfall model, date of acquisition 5 April 2011). It is also the oldest software development model of the traditional development models (Khalaf & Al-Jedaiah 2008, 1970). It is a sequential model where each phase is completed before next one is started, e.g. design must be fully completed before the implementation can start. This is illustrated in Figure 1.

Waterfall model, which name derives from a diagram that shows progress flowing down from one phase to the next is also a process that promotes copious documentation, rigid up-front definitions of requirements and system design, and division of a project into serialized phases. (Langr 2005, 9.)

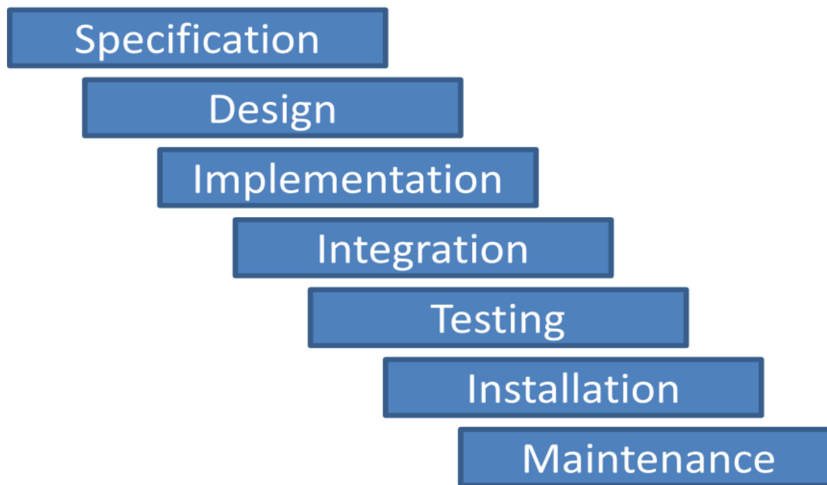

Figure 1. Usual phases in the waterfall development model.

Also, the output from each phase is used as an input in the next process. E.g. the output of specification phase is specification document that is used as the input document in design phase. Although not visible in the figure above, each phase also has a validation and verification phase, where the output of the specific phase is matched against its requirements. (Waterfall model, date of acquisition 5 April 2011.)

This kind of strict development model also has weaknesses. For example, if some fatal flaws in the software interfaces are spotted in the software product as late as in testing phase, it might be very difficult to go back to design phase to redesign the interfaces. After all, there might very high amount of software components affected. Also, bugs tend to be cumulative: the more there are bugs, the harder it is to remove each one. (Fowler 2006.)

Also, some software professionals think that is almost impossible to finish each phase of the model perfectly before advancing into next phase. For example, the customers might not know all requirements before they have seen a working prototype. They might also change their requirements when the development project advances. (Understanding the pros and cons of the Waterfall Model of software development, date of acquisition 5 April 2011.)

Langr also confirms this by saying that one of the key limitations in the waterfall model is that the project using it is less able to adapt to the changes in requirements. For this reason waterfall model is sometimes referred as heavyweight process. (2005, 10.)

2.3 Agile methods

Agile refers to a group of software development methods based on the same principles, e.g. Scrum and XP. Agile manifesto was signed by 17 software professionals in 2001, who believed in more lightweight software development model than classic waterfall. (Manifesto for Agile Software Development, date of acquisition 7 April 2011.)

The values of agile methods as described in agile manifesto are:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Also, the agile process follows 12 principles listed in agile manifesto, which are seen below (Manifesto for Agile Software Development, date of acquisition 7 April 2011):

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity, the art of maximizing the amount of work not done is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

2.4 Test-Driven Development

Agile methods usually also employ TDD that is originally derived from XP, which in turn belongs to family of agile methods. TDD is a technique where system is specified in term of tests. In other words, unit test cases are written prior to or as the new code is written. (Langr 2005, 18.)

TDD is a simple, short-cycled mechanism that is repeated during the implementation. According to Langr (2005, 19) its development cycle consists of following steps:

- Specification is written, in code and in form of a unit test.
- Test failure is demonstrated.

- Code is implemented to meet the specifications.
- Test success is demonstrated.
- Code is refactored to ensure clean and optimal code base.

When each test is executed against the entire system at all times, it ensures that no new code breaks the existing functionality or anything else in the system. TDD brings following positive aspects in the software development:

- Quality is improved, because TDD minimizes the number of defects, since by definition, everything is tested in the system. The design of the system is also improved, because TDD drives the development to the direction where classes are more decoupled from each other. Design where classes are not as heavily dependent on other classes are easier to test.
- Each unit test specifies the appropriate use of a production class, thus documenting its capabilities.
- Because of extensive use of unit tests, code can be improved and optimized without fear of breaking something that already works. This can also bring down the maintenance costs of software.
- As each cycle in TDD is very short, the feedback is provided quickly. Thus, developer discovers quickly if he is going into wrong direction in the implementation.

(Langr 2005, 19)

3 CONTINUOUS INTEGRATION

Fowler writes that back in the early days of software industry, one of the most troublesome moments of a software project was the integration. It was the phase where all modules that worked individually were finally put together producing a system that usually failed in ways that were difficult to find. (Duvall et al. 2008.)

The CI is a practice in agile methods to continuously integrate small code increments or commits automatically, thus avoiding the late big bang integration mostly present in traditional development models, like waterfall model (Fowler 2006).

3.1 Developer's workflow

At minimum the CI process implements the automated SW builds. However, usually the process also employs automated unit tests, QA metrics and reporting (Fowler 2006). Example of high-level CI setup is illustrated in figure 2.

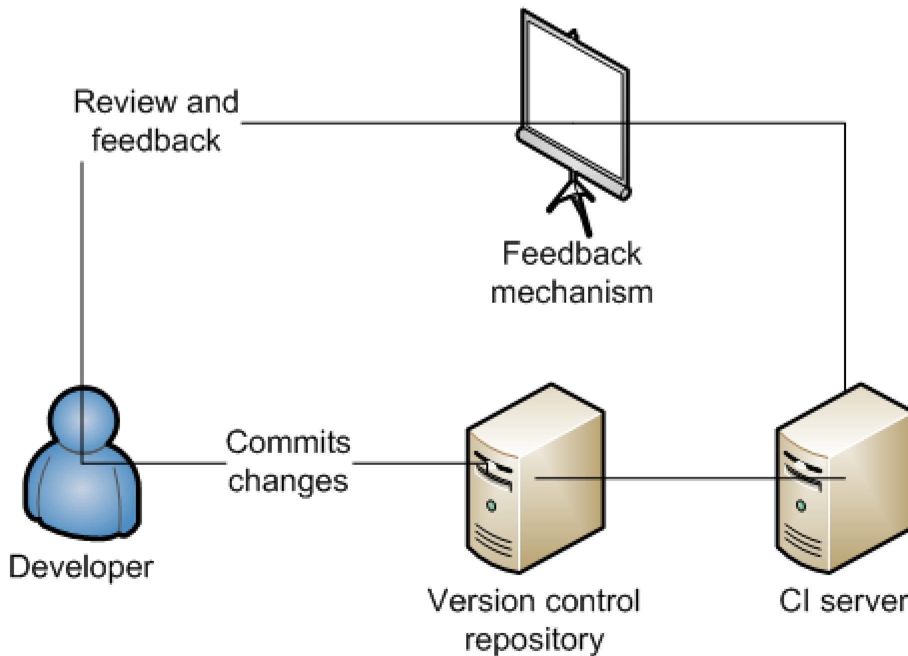

Figure 2. Simple CI setup.

Developer commits his newly implemented change into the repository. Few seconds later the CI server discovers the new change, pulls it into the CI server and starts the build. The commit is also reviewed by some automated tool or review requests sent to the other developers. After the build has finished, feedback (e.g. email notification) is sent back to developer. Also, if the commit was verified to be good by the CI server and the review was passed, it is automatically merged into the master repository. (Duvall et al. 2008, 5). Let us consider following two fictional, but quite realistic development scenarios:

Scenario 1: Traditional software development

> *Developer A should start working on a new software component that depends on some lower level library routines. However, developer B, who has been developing the library routines, has not released his changes yet. The changes have been completed, but not integrated into the master repository yet, because the integration team only integrates once a week. Thus, developer A needs to wait several days before the changes to the library routines are available for him to use via software release made by the release team.*

Scenario 2: Development method employing continuous integration

> *Developer B completes his new implementation into the library code and commits his changes into the staging repository. The CI server automatically builds and runs test suite for the new commit and upon successful result merges the commit into the master repository. Developer B then notifies developer A who can start developing his software component right away. This kind of integration cycle can easily happen several times a day.*

As seen from the example scenarios, integrating frequently can speed up the development process. The better quality compared to traditional models come in a form of extensive use of automated testing and continuous feedback, both by automated tools and peer reviewers. (Duvall et al. 2008, 27.)

3.2 Software releasing

James Shore and Shane Warden write (2008): "When you integrate continuously, releases are a painless event. Your team experiences fewer integration conflicts and confusing integration bugs. The on-site customers see progress in the form of working code as the integration progresses."

When the software project is employing continuous integration in their working practices, the software product can be released at any given time. CI builds ensure that the builds are never broken. Automated quality metrics tools and peer code reviews ensure that the actual software quality is at the required level. Thus, the software stabilization period prior to product releasing will decrease, because integration and releasing teams do not have to hunt for late bug fixes. (Duvall et al. 2008, 191.)

3.3 Benefits of continuous integration for software project

According to Duvall (Duvall et al. 2008, 29), the high level value of CI is to:

- Reduce risks
- Reduce repetitive manual process
- Generate deployable software at any time and at any place
- Enable better project visibility
- Establish greater confidence in the software product from the development team

Fowler (2006) confirms this by writing that with CI the project does not just reduce risks introduced in the long-running integration phase, but it actually eliminates the whole phase. The project also eliminates a lot of predictability problems, because usually it is very difficult to know how long the integration phase takes.

Regarding the bugs introduced in the implementation phase, Fowler continues by saying that CI does not help to get rid of bugs completely, but it makes them a lot easier to find and remove. Also, bugs might be cumulative and thus very difficult to get rid of in the later phase. (2006.)

Also, CI enables project to practice continuous deployment. Frequent deployment is important, because it allows users to get more features rapidly and also enables early feedback from users to developers. Thus, it helps to break barriers between development and customers. (Fowler 2006.)

4 JENKINS CI SERVER

This chapter describes the basics of Jenkins CI server. It covers the project background, installation, configuration and usage of the software.

Advanced features are also briefly explained. These features are CI server scalability and extending of Jenkins.

4.1 About Jenkins

Jenkins is an open source CI server based on Java platform. The purpose of Jenkins is to offer a framework for executing software builds and to run reporting and monitoring jobs. (Jenkins CI, date of acquisition 7 Feb 2012.)

Jenkins can be easily extended by using its extension API and plug-ins. There is already wide array of open source plug-ins available freely in the internet. (Jenkins CI, date of acquisition 7 Feb 2012.)

4.2 Project background

The development of Jenkins, or Hudson as it was originally called, was started by Kohsuke Kawaguchi as a personal hobby project late 2004 when working at Sun Microsystems. Soon other developers joined the project and development pace was fast. (Smart 2012, 4.)

In 2009, Oracle bought Sun Microsystems and tension between Oracle and Hudson developers rose e.g. about different opinions of the project management. In the beginning of January 2011 the source code of Hudson was forked and migrated to GitHug servers as Jenkins. (Smart 2012, 4.)

4.3 Distribution and installation

Jenkins is an open source software project and its source code and binaries are freely available in the internet. In its most simple form Jenkins is shipped in single WAR file that can be launched with Java SDK. Thus, platforms that have Java support also can run Jenkins, e.g. Windows, various Linux distributions, BSD distributions and Solaris (Smart 2011, 48-59). There are also native software packages for some of the platforms (Jenkins CI, date of acquisition 7 Feb 2012).

The release schedule of Jenkins is quite fast-paced as the project is releasing new version of Jenkins with bug fixes and new features every week. However, they also have long-time support version of Jenkins distributed for users who do not want to upgrade to new version as often. The latter could be very useful in e.g. corporate environment where software upgrades must happen in more controlled way. (Jenkins CI, date of acquisition 7 Feb 2012.)

4.4 Configuration and integration into enterprise systems

The configuration interface of Jenkins is mostly web based (Smart 2011, 17); only some Java parameters like virtual memory allocation must be configured before Jenkins server is started (figure 3). Naturally also disk space must be allocated for all generated files (Smart 2011, 47).

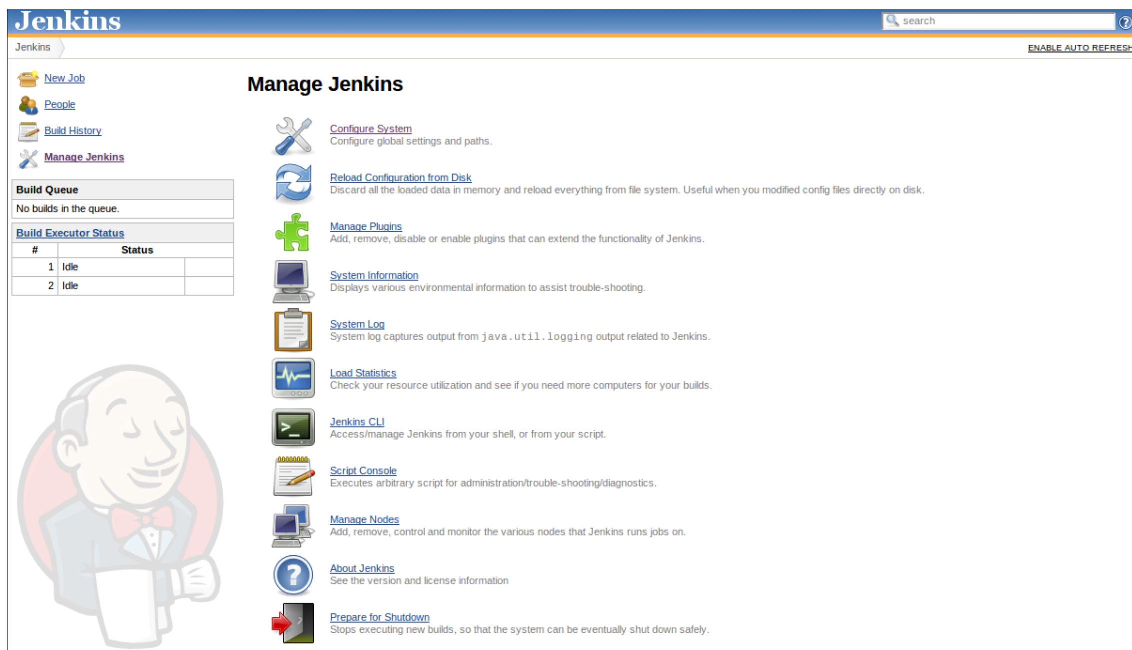

Figure 3. Jenkins server management view.

When Jenkins is started all its settings are in their default values. Usually first thing to configure is the user authentication. Jenkins supports several authentication mechanisms like local accounts, but usually LDAP is used in the enterprise networks (figure 4). By using LDAP all users who can access the network domain also can access Jenkins. The definition of access here is that they can create, modify, delete and run jobs plus make changes into server configuration, but access to e.g. server configuration can be restricted if so wanted and only allowed for server administrators. (Smart 2011, 171-182.)

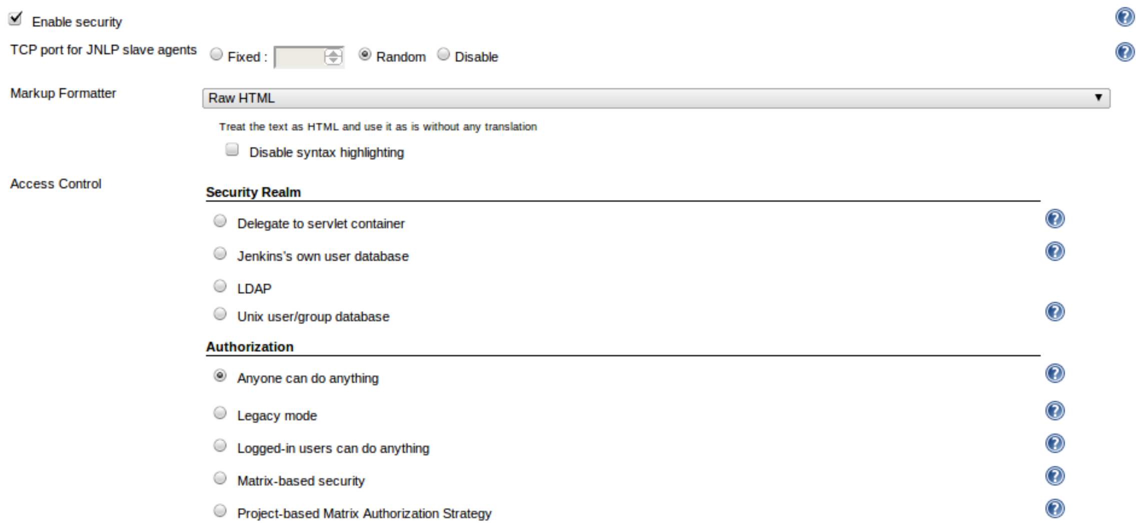

Figure 4. Jenkins security settings.

Other things to configure before production use are e.g. email server information and version control system. Although most of the version control system parameters are controlled in the user jobs there are also some global options for the whole Jenkins server instance. Also, the administrator can choose to install or upgrade some plug-ins that are not shipped in the default distribution of Jenkins. (Smart 2011, 71.)

4.5 User interface

As said earlier in this chapter, Jenkins employs a very easy to use web interface for running and managing jobs and system configuration. Figure 5 represents the UI in its default state right after installation.

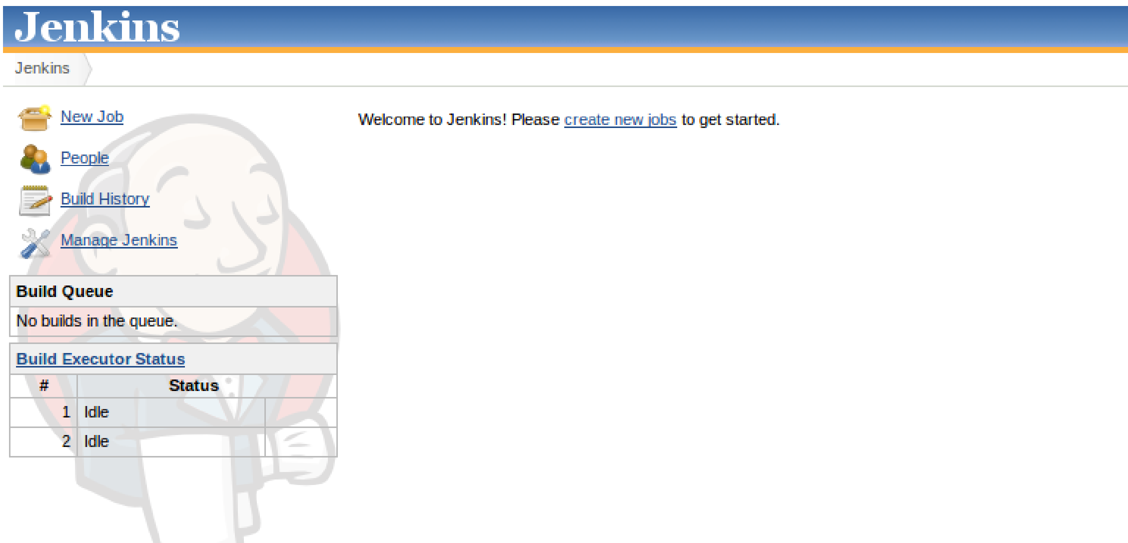

Figure 5. Jenkins web interface after installation

The quick links to main features of Jenkins are available in the upper left corner whereas the lower left corner displays the job queue and status of executors. The executors can be thought as parallel processes and the amount of them can be configured from the Jenkins management view. Generally it's a good practice to have at least as many executors as there are CPUs in the server. The optimal amount of course depends on the usage pattern of server. (Smart 2011, 349.)

New plug-ins and upgrades to existing plug-ins can be installed straight from the web UI. This makes maintenance of Jenkins server very simple.

4.6 Job types

Jenkins supports different kind of jobs. Each job type has different characteristics and can be used to do some specific thing.

Free-style jobs are very flexible general purpose job types. They can be used with any SCM and build system. Also, free-style jobs can be used for other purposes than software builds. (Smart 2011, 82-84.)

Maven job type can be used with projects which are using Maven build management system. Apache Maven is a software and build management tool used widely in Java development. It is based on a concept of project object model (Apache Maven, date of acquisition 7 Feb 2012). Jenkins can take advantage of the project’s POM file and thus these jobs require only very little configuration (Smart 2011, 118-127).

Multi-configuration jobs can be used with projects that need a large number of different configurations. A good example of multi-configuration job could be the case where the same software project needs to be built for several platforms or environments. (Smart 2011, 268-273.)

It is also possible to monitor external processes with Jenkins. This can be accomplished by using a specific job type designed for this purpose. Thus, it is possible to hook external processes into CI server and use Jenkins as a dashboard. (Smart 2011, 351.)

4.7 Extending Jenkins

Jenkins can be extended in several ways. As Jenkins is an open source project, the developer may choose to participate into its development and contribute code or documentation directly into Jenkins core. (Jenkins CI, date of acquisition 7 Feb 2012.)

It is also possible to participate into development by improving existing plug-ins or developing new ones. There is lot of documentation in the internet about how to develop plug-ins. (Extend Jenkins, date of data acquisition 2 June 2012.)

4.7.1 Built-in components

As noted in the beginning of this chapter, Jenkins can be easily extended and more features implemented by taking advantage of its extension architecture. Jenkins defines so called extension points, which are interfaces or abstract classes that model the concepts of build system in a straight-forward fashion. These interfaces define contracts that need to be implemented in order to contribute to Jenkins. An example of such an interface is SCM interface. In order to add support for new SCM system, the new extension must implement SCM interface. (Kuchana 2004, 24.)

The benefit of using interfaces is that the client component does not need to be modified if the service provider (i.e. class that implements the interface) changes or even when a new service provider is designed as part of the class hierarchy (Kuchana 2004, 24).

4.7.2 Plug-ins

In addition to built-in extension points and their implementation, Jenkins architecture also supports notion of plug-ins, which are dynamically loadable modules that implement certain additional functionality. The plug-in can also plug into Jenkins extension points, provided they implement certain interfaces. Each plug-in is loaded into separate class loaded to avoid conflicts with built-in classes and other plug-ins. From the user point of view the plug-ins are as integrated into Jenkins as its built-in features. It is also possible to enable, disable, upgrade and install plug-ins via Jenkins web UI (figure 6). (Smart 2011, 278.)

Updates | **Available** | Installed | Advanced

Install ↓	Name	Version
Artifact Uploaders		
	Appaloosa Plugin Publish your mobile applications (Android, iOS, ...) to the appaloosa-store.com platform.	1.3.0
	ArtifactDeployer Plugin This plugin makes it possible to copy artifacts to remote locations.	0.16
	Artifactory Plugin This plugin allows deploying Maven 2, Maven 3, Ivy and Gradle artifacts and build info to the Artifactory artifacts manager.	2.1.0
	Backlog Plugin This plugin integrates Backlog to Jenkins.	1.7
	Build Publisher Plugin This plugin allows records from one Jenkins to be published on another Jenkins.	1.12
	Confluence Publisher Plugin This plugin allows you to publish build artifacts as attachments to an Atlassian Confluence wiki page.	1.5
	Deploy Plugin This plugin takes a war/ear file and deploys that to a running remote application server at the end of a build	1.8
	Deploy WebSphere Plugin This plugin is an extension of the Deploy Plugin. It takes a war/ear file and deploys that to a running remote WebSphere Application Server at the end of a build.	1.0
	Dimensions Plugin This plugin integrates Hudson with Dimensions, the Serena SCM solution.	0.8.1

Figure 6. Jenkins plug-in manager.

4.8 Scalability

Jenkins can also run in a master-slave setup where e.g. build jobs from master node are automatically distributed for execution to the pool of slave nodes, which increases the throughput of CI system by distributing the CPU and I/O load across the whole pool of machines. New slaves can be added dynamically without restarting the server process. (Smart 2011, 306-307.)

Also, it is possible to mix different OS platforms in the pool which means the pool could have e.g. slaves with Linux, Windows and Solaris operating system for testing software builds on different platforms and environments. (Smart 2011, 306-315.) Example of slave being configured for Solaris OS is illustrated in figure 7.

Name Solaris_slave

Description Slave for running SW builds on Solaris

# of executors 4

Remote FS root /

Labels

Usage Utilize this slave as much as possible

Launch method Launch slave agents on Unix machines via SSH

Host

Advanced...

Availability Keep this slave on-line as much as possible

Node Properties

Environment variables

Tool Locations

Save

Figure 7. Configuring Jenkins slave node for running builds on Solaris.

5 DEVELOPMENT

This chapter describes the requirements for the black box plug-in, the development schedule and the design principles. It also illustrates how the software was implemented, tested and finally deployed into the commissoner's CI server.

Besides the author of this thesis work, there were also other stakeholders that participated into the development of the plug-in. Integration, releasing and delivery chain managers gave input about the requirements of plug-in whereas the SW supplier gave valuable feedback throughout the whole development process. The CI project was responsible of maintaining the CI infrastructure and also releasing the plug-in updates.

5.1 Requirements

The following basic plug-in requirements were identified by the commissioner, because they were seen as critical items in the commissioner's software delivery chain and development process where external SW suppliers are used:

a) The plug-in must be able to download SW supplier's source code using FTP, FTPS or FTPES from remote server. The source code can be supplied either as a plain directory hierarchy or a ZIP file.
b) The plug-in must be able to interface with VCS to check out commissioner's source code.
c) The plug-in must be able to build SW supplier's source code with source code from the commissioner.
d) The plug-in must be able to store and upload build artifacts, logs and reports back to partner company by using FTP.
e) The plug-in must be able to send email notifications and reports about the build statuses to predefined list of email addresses.

f) The plug-in must be written to comply with commissioner's coding style standards for good maintainability and extensibility.

5.2 Schedule

The design and implementation schedule was dictated mostly by the business needs of commissioner. The total workload of thesis work was also aligned to match requirements as defined by the degree programme of Oulu University of Applied Sciences. The actual project work was carried out as shown below.

- Weeks 11-14/2011
    - Discussions with commissioner representative about development task and its constraints
    - Background research (literature, published research and articles, internet)
    - Getting familiar with development tools and environment
    - Architectural planning
    - Requirement specifications as input from commissioner
- Week 15/2011
    - Idea seminar 13.4.2011
    - Design and implementation begins
    - Thesis writing
- Weeks 16-28/2011
    - Implementation and continuous testing
    - Verification and validation
    - Thesis writing
- Weeks 28-32/2011
    - Pilot use with selected partner company (weeks 28-32)
    - Feedback from commissioner
- Weeks 32-40/2011
    - Deployment into production use (week 34)
    - Thesis writing

- Weeks 41/2011-19/2012
    - Thesis writing
    - Planning seminar
- Weeks 20-22/2012
    - Thesis writing, finalization and binding
    - Publishing seminar

It is worth to note that the original schedule was much tighter and the thesis was supposed to be ready in October 2011. However, due to issues not related to this thesis work, the writing of thesis was delayed by several months. However, the actual software design, implementation, testing and deployment were completed in schedule.

5.3 Design

Jenkins CI server and its APIs have been developed with Java language so the plug-in was developed using the same language and design principles (Jenkins API documentation, date of acquisition 17 April 2012).

In order to achieve a clean code base, easier maintainability and good modular structure of the software component, the plug-in code was placed into several Java classes based on the functionality (table 1):

- Networking
    - FTP connection handling
    - Downloading
    - Uploading
- SCM operations
    - Checking out source code
    - Checking in source code
    - Committing source code into the VCS
    - Publishing (i.e. releasing) source code

- Email handling
- Data models
- Configuration handling

Table 1. Classes that implement black box plug-in.

Class summary	
BlackboxBuilder	Implements main class of the plug-in.
BlackboxBuilder.DescriptorImpl	Implements plug-in descriptor for Jenkins.
BlackboxConfiguration	Stores Jenkins job configuration.
BlackboxFTPConnector	Establishes/disconnects FTP sessions.
BlackboxLogger	Implements different logging levels for the plug-in.
BuildInfo	Stores persistent build data for Jenkins UI.
BuildResult	Data model to stores a result of SW build.
ConfigurationParser	Parser for XML configuration files.
CredentialHandler	Encrypts and decrypts passwords.
EmailNotifier	Handles generation and sending of email messages.
FileDownloader	Implements easy way to download files and directory hierarchies from the FTP server. Uses FTP4J class library.
FileUploader	Implements easy way to upload files to the FTP server. Uses FTP4J class library.
GlobMatcher	Helper class for file pattern matching. File patterns can be utilized in the artifact upload phase.
ScmOperations	Implements an API towards version control system.

High level architectural view of black box plug-in design gives a good overview of the functionality of plug-in (figure 8). Each step is explained in more detail later in this chapter.

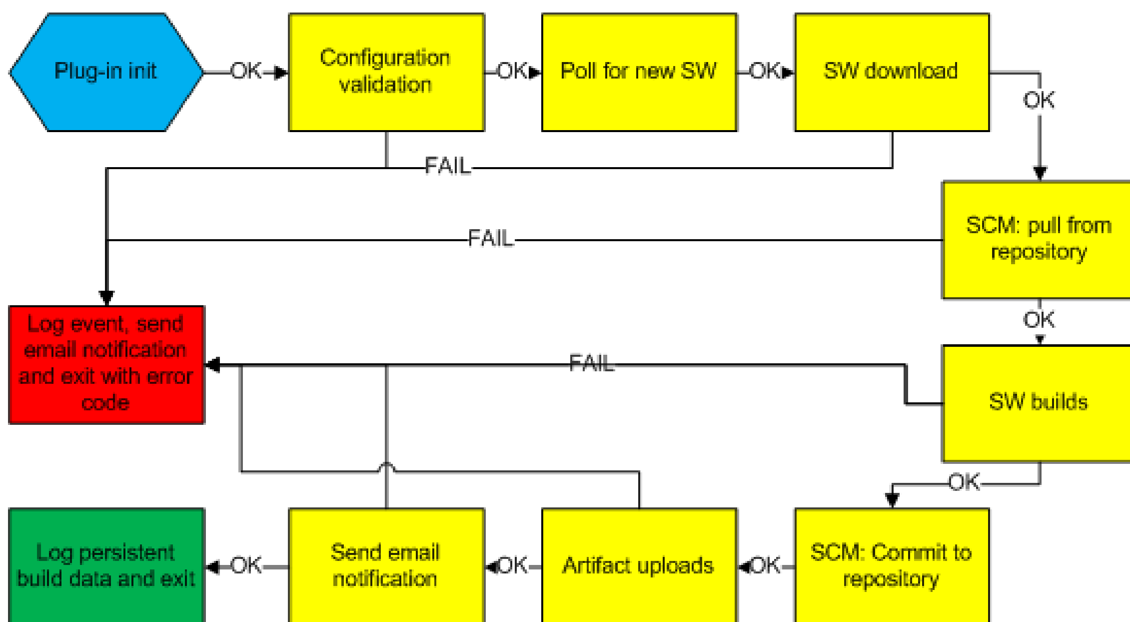

Figure 8. High level architectural view of black box plug-in.

It is generally considered a good practice to avoid reinventing the wheel. Thus also the black box plug-in employs some 3rd party Java class libraries (table 2).

Table 2. 3rd party class libraries used by the black box plug-in.

3rd party class libraries		
Library name	Used by	License
commons-httpclient	DependencyTaskHandler class for making HTTP connections.	Apache Software License
commons-codec	CredentialHandler class for encrypting/decrypting data.	Apache Software License, Version 2.0
FTP4J	Several classes for FTP/FTPS/FTPES connections.	LGPL

5.3.1 Plug-in init

When the Jenkins job containing the black box plug-in is started, Jenkins creates a new instance of BlackboxBuilder class which is the main class of black box plug-in. The plug-in then initializes its data structures, logger and other vital parts to its operation.

Also some sanity checks about the environment are made. For example, the existence of external tools and their required versions are checked at plug-in initialization.

5.3.2 Configuration validation

The plug-in has server side configuration for SCM parameters, FTP connection parameters, administrator email addresses and location of client side configuration file. These configuration options are stored within the Jenkins job. (Figure 9)

CI Configuration file	/path/to/the/configuration/file.xml
Directory to replace in work area	path/to/subdirectory
SCM job name	Blackbox_SCM
Build job name	Blackbox_build
Number of builds to keep	5
Synergy database path	/path/to/synergy/database
☑ FTP configuration	
FTP server host	ftp.somesite.com
FTP server port	990
Username	username
Password	••••••••
	Test Connection
☑ Email configuration	
Sender address	admin@localhost
Admin/BCC recipients	
Email subject prefix	[JENKINS] Blackbox
Email body prefix	

Figure 9. Configuration view of the black box plug-in.

When the plug-in is activated by Jenkins it first validates the server side parameters and then proceeds to download the client side configuration file. Figure 10 illustrates the process. The client side configuration parameters are then read from the file and validated. The client side configuration consists of:

- List of email addresses where the notifications will be sent
- Location of input data (zip or directory hierarchy)
- Location of output data where the build artifacts and logs will be uploaded

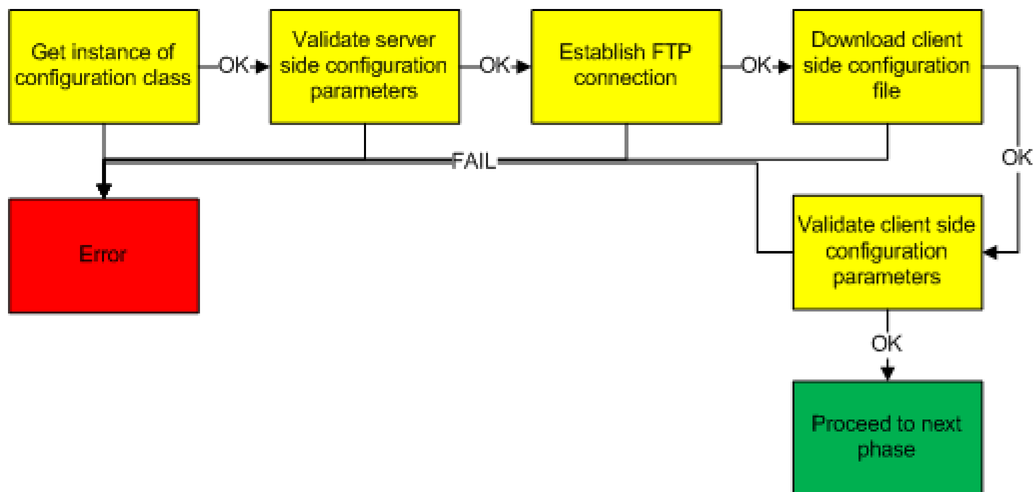

Figure 10. Process flow of configuration validation.

If client side configuration parameters are not valid or configuration file cannot be found, the plug-in logs the error in Jenkins console and returns with error code. Example of plug-in run where client side configuration file cannot be downloaded can be seen in figure 11.

Jenkins » Blackbox » #1

- Back to Project
- Status
- Changes
- Console Output
- Edit Build Information
- Build information

Logger Console Expand

Console Output

```
Started by user anonymous
Building in workspace /home/mikvaana/plugin_dev/s40-ci-blackbox/work/workspace/Blackbox
[I][2012-05-02-20:18] Disabling project to prevent execution queue from filling up
================================================================================
          BLACK BOX (version 1.0.6)
--------------------------------------------------------------------------------
[I][2012-05-02-20:18] Checking if SCM job 'Blackbox_SCM' is running
[E][2012-05-02-20:18] Cannot find job with name 'Blackbox_SCM'
[I][2012-05-02-20:18] Checking if build job 'Blackbox_build' is running
[E][2012-05-02-20:18] Cannot find job with name 'Blackbox_build'
[E][2012-05-02-20:18] Could not get configuration file: /path/to/the/configuration/file.xml
[E][2012-05-02-20:18] Initial checks failed
[I][2012-05-02-20:18] Inserted build data successfully
[I][2012-05-02-20:18] S40 Black box finished
[I][2012-05-02-20:18] Enabling project
Build step '          Black box' marked build as failure
Finished: FAILURE
```

Figure 11. Example of failed black box run.

5.3.3 Poll for new SW

Due to firewall configuration issues the SW supplier cannot necessarily trigger the Jenkins jobs by themselves or via some script. Therefore, the plug-in must poll for new SW at predefined intervals (figure 12). After the client side configuration file has been processed, the plug-in connects to remote FTP server and looks for new SW from the input directory. If no new SW can be found, the plug-in logs the event and exists with non-zero return code.

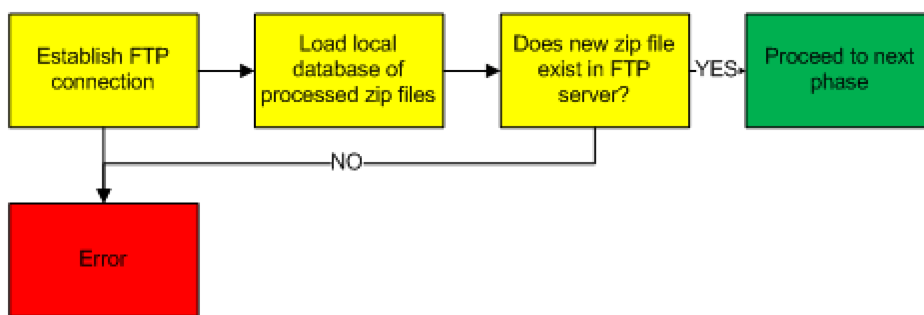

Figure 12. Process flow of new zip file detection.

The plug-in keeps track of already processed zip files in a local database.

5.3.4 SW download

If new SW exists in the input directory the plug-in downloads it into Jenkins server with FTP. If the download fails (e.g. because of FTP connection problem) the plug-in logs the event into Jenkins console and exists with return code.

When the download begins the plug-in starts the next phase in other thread. This is because both phases are time-consuming and independent from each other.

5.3.5 SCM: Pull from repository

The plug-in starts another Jenkins job in which the actual SCM plug-in has been activated. It connects to source code repository and updates the working directory with newest source code.

If the SCM operation fails (e.g. SCM update is incomplete and thus working directory content is not valid) the SCM plug-in logs the event and exists with error code.

5.3.6 SW builds

The plug-in started another Jenkins job in which the actual SW builds happen. The builds can be scripted or there may be some kind of build plug-in. The build phase can consist of one or more SW builds. (Figure 13)

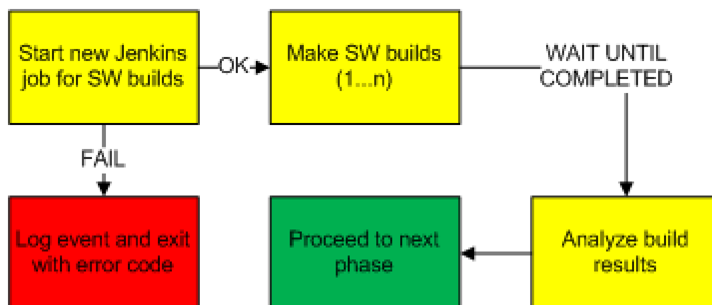

Figure 13. Process flow of making software builds.

The black box plug-in waits until all SW builds have completed and build job terminated. After that build job is queried for results which are stored in objects derived from BuildResults class.

5.3.7 SCM: Commit to repository

If SW builds were successful the black box plug-in will commit the downloaded SW into the source code repository if it was configured so. The SW can then be retrieved from the source code repository by the integration and releasing teams.

All commits into source code repository are tagged with a unique identifier. This way the commits in the repository can be easily traced back into correct black box build event.

5.3.8 Artifact uploads

The build artifacts will be uploaded into the output directory that was specified in the client side configuration files. To reduce the needed bandwidth and time required for the data transfer, the artifacts will be compressed before FTP transfers. Build artifacts can be e.g.:

- SW images, program executables, object files and libraries
- Build logs
- SW quality reports by automated analysis tools
- SW test reports by automated testing tools

5.3.9 Email notification

Finally, the plug-in will generate email message about its run. The email will be sent to predefined set of administrator addresses plus addresses which were configured in the client side configuration file. Following information will be provided in the message:

- SW version to identify input data
- SCM ID

- Build results
- Location of build artifacts
- HTTP link into Jenkins job
- Version and build date of black box plug-in

5.4 Implementation and testing

This chapter covers the details about the development environment. The environment, which included a lot of different tools, was fully based on the open source software.

Also, the plug-in development process is briefly explained. Agile methods and CI were also employed in the development of black box plug-in.

5.4.1 Agile development method

The implementation of plug-in started in April 2011. The focus was to get absolutely minimum functionality implemented first and then implement the rest of features on top of that. The following minimum features were identified by the commissioner:

- Ability to download SW supplier's source code from the FTP server
- Ability to make SW build
- Basic email notifications for build results

After the minimum requirements were implemented, test version of the plug-in was deployed into the CI server for pilot use in production environment. When bugs were removed and new features developed, newest version of the plug-in was always installed in the CI server. Thus, agile development methods were heavily employed in form of continuous integration, testing and deployment.

5.4.2 Development environment

Although Jenkins is well supported in several platforms, the commissioner is using mostly Linux based servers. Thus, it was very natural choice to use Linux also as a development platform. The development environment consisted of:

- 64-bit Linux OS
- IntelliJ IDEA IDE
- Java SDK
- Apache Maven
- Jenkins and its dependencies

Java is a technology and software framework consisting of the Java programming language, class libraries and virtual machine to run Java bytecode. It is used heavily in wide array of software projects. (Java, date of acquisition 1 Feb 2012.)

Jenkins is using Apache Maven as software and build management solution and thus it was also used in the plug-in development. Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. (Apache Maven, date of acquisition 7 Feb 2012.)

IntelliJ IDEA is a well-supported code-centric IDE focused on developer productivity. It has very powerful source code editor with code completion, easy to use refactoring functionality, code navigation and integrated debugger. It also has a very good support for Maven based projects. The free community version of the IDE was used in the plug-in development, but there is also a commercial version available. Its well-organized window layout is shown in figure 14. (Intellij IDEA, date of acquisition 7 Feb 2012.)

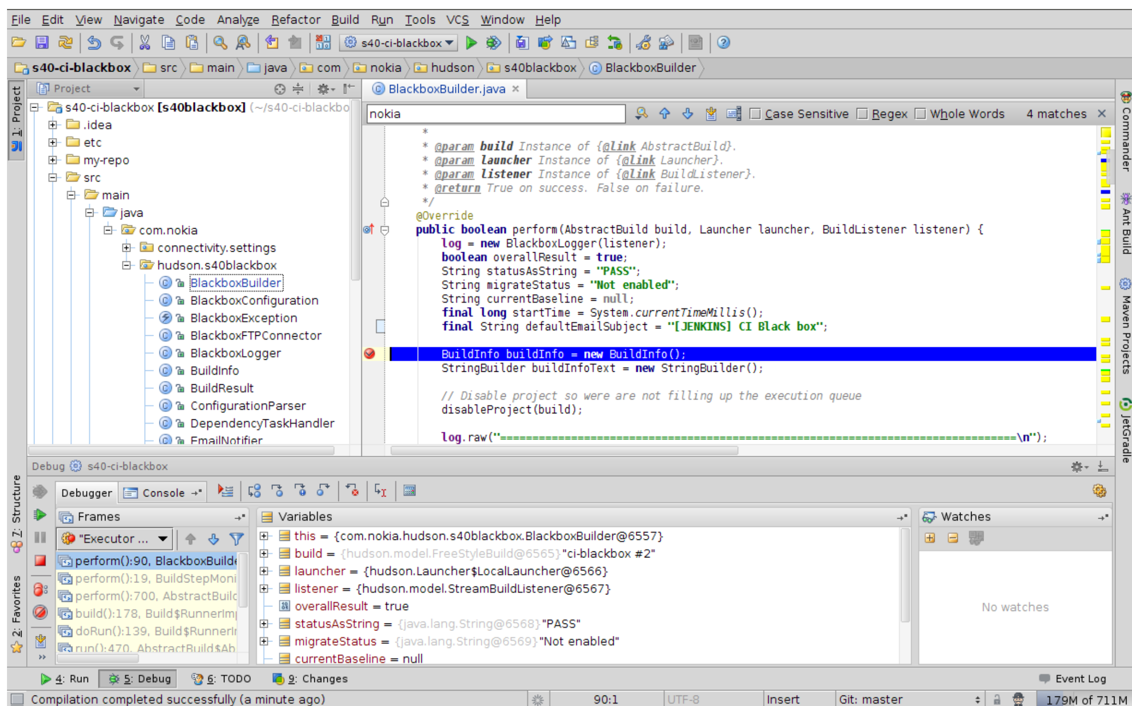

Figure 14. Debugging view of IntelliJ IDEA.

The plug-in was developed locally on the computer running Linux, but testing was mostly performed on commissioner’s CI server running company-customized version of Jenkins.

Bugzilla (Bugzilla, date of acquisition 7 Feb 2012), that is heavily used in the open source development was used for the error management of plug-in and git (Git, date of acquisition 7 Feb 2012) for source code management. Also, CI was employed during the development of this plug-in meaning that every commit was verified and tested by CI system.

5.5 Deployment

The initial deployment of plug-in for SW supplier's use was performed earlier than originally planned, already in June 2011. Separate Jenkins instance was started on the CI server and the plug-in was installed there.

As more features were implemented and bugs were removed, the newest version of plug-in was always deployed into the CI server to get early feedback from the SW supplier and all other stakeholders. Also Jenkins was updated at the same time if newer production-ready version was available. When the plug-in was stable enough, it was integrated into commissioner's internal Jenkins release for easier deployment. The plug-in was distributed into several development sites across the globe by using customized Jenkins releases.

6 CONCLUSIONS AND DISCUSSION

This chapter describes the conclusions of project. Also personal opinions of the author about the development process are shared.

Future development possibilities and ideas are also covered briefly. The maintenance and development of black box plug-in has been transferred to the CI project of commissioner and more features are implemented when needed.

6.1 About the thesis process

I think this was a very challenging project for me. I did not have previous experience of Java platform and I have only used Jenkins as an end-user before, and not very much of that either. Also, when working in the software integration team I have developed some tools to assist other integrators and technical employees in their daily work, but this was by far the biggest software product that I have been responsible of. On the other hand, I was quite familiar with the software development and its processes so learning curve for new programming languages and techniques was not that steep.

The original plan was to finish the thesis work by October 2011. There were several factors that contributed into the delay, but mostly it was because of some other high-priority tasks I needed to do first in my daily work. Also, there were some new requirements during the implementation in the quite late phase. However, this is quite normal in agile development where features are implemented and released incrementally and new requirements might pop up all the time. The overall object-oriented design of the plug-in allowed for easy implementation of new features.

The feedback from the commissioner and SW supplier has been very good and I consider the planning, design, implementation, testing and deployment

successfully executed. The plug-in is now a very critical component in the SW delivery flow between the SW supplier and commissioner. Also, the plug-in does its job pretty much automatically and requires only a very little maintenance efforts. Thus, the project also brought very much value for the commissioner.

I have contributed a lot into different software components during my career, but this was my first software development project where I was the only developer and responsible of full end-to-end solution of such a critical component of the system. Thus, I learned valuable skills in the software product management, but also in some technology areas of which I did not have previous experience, such as Java language, its class libraries and naturally Jenkins framework.

All in all, the making of this thesis work was very important step for me in learning new skills and technologies and will surely help me in my career path. Also, the support from various stakeholders during the project has been tremendous.

I would like to express my greatest gratitude to my loved one, Hanna, for her endless patience, encouragement and support in my thesis process. She gave me strength and hope at times when I needed them most. Thank you!

6.2 Future development plans

There are already some plans to develop the black box plug-in further, to implement more features and adapt to changes in development environment.

One of the key features of future development is the support for more version control systems. Although the black box plug-in was designed to be as generic as possible, different version control systems do have different interfaces and some are also fundamentally very different from each other. Thus, some changes to plug-in design are also needed.

Also, the plug-in was originally designed to work in standalone Jenkins installation. Nowadays the commissioner is using master-slave setup quite heavily so the plug-in must also adapt into that setup.

There has also been some discussion about using some kind of binary storage system to store all input and output data instead of FTP server. Such an interface could be implemented to the plug-in very easily.

The maintenance and development of the plug-in has been moved into the CI project of commissioner. I will still continue to provide support and guidance when needed.

6.3 Documentation

The plug-in classes and methods were documented into the source code using standard Javadoc notation. This way it is easy to generate consistent HTML documentation about the project at any given time without needing to update any documents in the separate document management system or repository. The up-to-date source code documentation is very valuable source of information to the developers who are going to maintain and develop the software component further. (How to Write Doc Comments for the Javadoc Tool, date of acquisition 2 June 2012.)

An installation and configuration document was also created for commissioner's internal use. It is mostly targeted to the Jenkins administrators who will maintain and update the black box plug-in configuration in the integration and releasing team.

REFERENCES

Apache Maven. Date of data acquisition 7 February 2012 http://maven.apache.org/

Bugzilla. Date of data acquisition 7 February 2012 http://www.bugzilla.org/

Duvall, P., Matyas, S. & Glover, A. 2008. Continuous Integration: Improving Software Quality and Reducing Risk. Boston: Pearson Education, Inc.

Extend Jenkins.Date of data acquisition 2 June 2012 https://wiki.jenkins-ci.org/display/JENKINS/Extend+Jenkins

Fowler, M. 2006. Continuous integration. Date of data acquisition 7 April 2011 http://martinfowler.com/articles/continuousIntegration.html

Git. Date of data acquisition: 7 February 2012 http://git-scm.com/

How to Write Doc Comments for the Javadoc Tool. Date of data acquisition 2 June 2012 http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

IntelliJ IDEA. Date of data acquisition 7 February 2012 http://www.jetbrains.com/idea/

Java. Date of data acquisition 1 February 2012 http://www.oracle.com/us/technologies/java/index.html

Jenkins CI. Date of data acquisition 7 February 2012 http://jenkins-ci.org
Jenkins API documentation Date of data acquisition 17 April 2012 http://javadoc.jenkins-ci.org/

Khalaf, S. & Al-Jedaiah, M. 2008. Software Quality and Assurance in Waterfall Model and XP - A Comparative Study. Date of data acquisition 7 April 2011 http://www.wseas.us/e-library/transactions/computers/2008/31-097.pdf

Kuchana, P. 2004. Software Architecture Design Patterns in Java. Florida: CRC Press LLC

Langr, J. 2005. Agile Java: Crafting Code with Test-Driven Development. New Jersey: Pearson Education Inc.

Manifesto for Agile Software Development. Date of data acquisition 7 April 2011 http://agilemanifesto.org/

Shore, J. & Warden, S. 2008. The Art of Agile Development. O'Reilly Media, Inc.

Smart, J. 2011. Jenkins: The Definitive Guide. Wellington: Wakaleo Consulting.

Understanding the pros and cons of the Waterfall Model of software development. 2006. Date of data acquisition 5 April 2011 http://www.techrepublic.com/article/understanding-the-pros-and-cons-of-the-waterfall-model-of-software-development/6118423

Waterfall model. Date of data acquisition 5 April 2011 http://courses.cs.vt.edu/csonline/SE/Lessons/Waterfall/