

# CONTINUOUS DEPLOYMENT WORKFLOW

Case Lego Mindstorms EV3

Janne Alatalo

Bachelor's thesis  
September 2014

Software Engineering  
The School of Technology, Communication and Transport



JYVÄSKYLÄN AMMATTIKORKEAKOULU  
JAMK UNIVERSITY OF APPLIED SCIENCES



Author(s) ALATALO, Janne	Type of publication Bachelor's Thesis	Date 04.09.2014
	Pages 60	Language English
		Permission for web publication ( X )
Title CONTINUOUS DEPLOYMENT WORKFLOW: Case Lego Mindstorms EV3		
Degree Programme Software Engineering		
Tutor(s) VÄÄNÄNEN, Olli		
Assigned by RINTAMÄKI, Marko		
Abstract <p>This thesis was created for N4S@JAMK project. The N4S@JAMK project is part of Need4Speed program run by DIGILE. The assignment for this thesis had two focuses: to create a continuous deployment chain that would deploy software to a Lego Mindstorms EV3 device, and to study ways to unit test the software of EV3 device on a normal computer.</p> <p>The continuous deployment chain was implemented using GitLab, Jenkins programs and Fabric python framework. Lego Mindstorms EV3 device had a third party firmware called MonoBrick installed instead of Lego's own firmware. MonoBrick firmware had a mono software included in the installation, which means the software developed for EV3 device was written in C#.</p> <p>The EV3 device's software was unit tested on a computer using a good software architecture and mock objects. The parts of the software using some hardware dependency, such as parts of software that accessed sensors of motors, were designed in a way that allowed to swap them to mock objects when running unit tests.</p> <p>The continuous deployment chain was tested with a test project and the implementation was presented in a demo bazaar event during Need4Speed program's Q2 review at Tampere.</p>		
Keywords Lego Mindstorms EV3, Continuous Deployment, Embedded Systems, Unit Testing		
Miscellaneous		



Tekijä(t) ALATALO, Janne	Julkaisun laji Opinnäytetyö	Päivämäärä 04.09.2014
	Sivumäärä 60	Julkaisun kieli Englanti
		Verkojulkaisulupa myönnetty ( X )
Työn nimi CONTINUOUS DEPLOYMENT WORKFLOW: Case Lego Mindstorms EV3		
Koulutusohjelma Ohjelmistotekniikka		
Työn ohjaaja(t) VÄÄNÄNEN, Olli		
Toimeksiantaja(t) RINTAMÄKI, Marko		
Tiivistelmä <p>Opinnäytetyön toimeksiantajana toimi N4S@JAMK projekti. N4S@JAMK on osa DIGILE-yrityksen vetämää Need4Speed-ohjelmaa. Opinnäytetyön painopisteenä oli kaksi eri aihetta. Ensimmäinen näistä aiheista oli jatkuvan julkaisun ketjun toteuttaminen Lego Mindstorms EV3 -laitteelle. Toinen aihealueista oli tutkia tapoja yksikkötestata EV3-laitteelle kehitettyjä ohjelmistoja tavallisella tietokoneella.</p> <p>Jatkuvan julkaisun ketju toteutettiin käyttäen GitLab- ja Jenkins-ohjelmistoja sekä Fabric python -kirjastoa. EV3-laitteessa käytettävä ohjelmisto oli kolmannen osapuolen tarjoama MonoBrick-laiteohjelmisto. EV3-laitteeseen esiasennettuna ollut Legon omaa laiteohjelmistoa ei käytetty. MonoBrick-ohjelmisto sisälsi monokehitysympäristön asennuksen yhteydessä, joten EV3-laitteelle kehitetty ohjelmisto kirjoitettiin C#-ohjelmointikielellä.</p> <p>EV3-laitteelle tarkoitettujen ohjelmiston yksikkötestaus toteutettiin hyvällä ohjelma-arkkitehtuurilla sekä mock-objekteilla. Ne ohjelman osat, jotka olivat riippuvaisia laitteistosta, kuten antureita ja moottoreita käsittelevät osat, suunniteltiin niin, että ne on helppo vaihtaa testauksen ajaksi mock-objekteihin.</p> <p>Jatkuvan julkaisun ketju testattiin testi-projektilla ja se esiteltiin demo bazaar -tapahtumassa Need4Speed-ohjelman Q2 Review -tilaisuudessa.</p>		
Avainsanat (asiasanat) Lego Mindstorms EV3, Jatkuva julkaisu, Sulautetut järjestelmät, Yksikkötestaus		
Muut tiedot		

# CONTENTS

<b>TERMINOLOGY .....</b>	<b>3</b>
<b>1 INTRODUCTION.....</b>	<b>5</b>
1.1 Objectives of this thesis.....	5
1.2 N4S@JAMK project.....	6
<b>2 CONTINUOUS DEPLOYMENT .....</b>	<b>7</b>
2.1 Definition of continuous deployment .....	7
2.2 Continuous deployment pipeline .....	10
2.3 Why continuous deployment .....	14
<b>3 SANDERO PRODUCTION ENVIRONMENT .....</b>	<b>15</b>
3.1 Sandero.....	15
3.2 OpenLDAP.....	16
3.3 Git .....	17
3.4 GitLab.....	18
3.5 Jenkins .....	20
3.6 Proxy.....	23
3.7 Fabric .....	24
<b>4 TEST AUTOMATION.....</b>	<b>25</b>
4.1 Test automation with embedded systems.....	25
4.2 Unit testing .....	27
4.3 NUnit and moq .....	29
4.4 Test-driven development .....	32
4.5 Automatic acceptance testing.....	33
<b>5 LEGO MINDSTORMS EV3.....</b>	<b>34</b>

5.1	Mindstorms EV3 brick.....	34
5.2	MonoBrick firmware.....	38
<b>6</b>	<b>ASSIGNMENT .....</b>	<b>40</b>
6.1	Goal.....	40
6.2	Starting point.....	41
6.3	Implementation .....	45
6.4	Demo .....	52
<b>7</b>	<b>CONCLUSION .....</b>	<b>56</b>
	<b>REFERENCES .....</b>	<b>58</b>

## FIGURES

FIGURE 1.	SANDERO ENVIRONMENT.....	16
FIGURE 2.	GITLAB WEB UI.....	19
FIGURE 3.	JENKINS WEB UI.....	20
FIGURE 4.	COLOROBOT DESIGN PATTERN .....	28
FIGURE 5.	THE LEGO MINDSTORMS EV3 PROGRAMMABLE BRICK .....	35
FIGURE 6.	SENSORS AND MOTORS COMING IN THE EV3 HOME SET.....	36
FIGURE 7.	SOME OF THE BUILDING BLOGS AVAILABLE IN THE EV3 HOME SET .....	37
FIGURE 8.	NODE MANAGEMENT WINDOW ON JENKINS .....	47
FIGURE 9.	FINAL CONTINUOUS DEPLOYMENT WORKFLOW.....	50
FIGURE 10.	COLOROBOT .....	53

## TABLES

TABLE 1.	JENKINS PLUGINS USED IN THE THESIS ASSIGNMENT.....	23
----------	--	----

## TERMINOLOGY

### **Mono**

Open source cross-platform implementation of the .NET framework developed by Microsoft.

### **Authentication**

Authentication is about verifying who the user is. This differs from authorization in that authorization says what the user can do.

### **A/B Testing**

Testing two variants of a program. With this kind of testing it is easy to test if users like for example some new feature by deploying two versions and asking for feedback.

### **CD**

Continuous deployment.

### **CI / CI Server**

Continuous integration / Continuous integration server.

### **C#**

Programming language developed by Microsoft.

### **Embedded system**

Computer system usually designed for a single purpose. It is said to be embedded since the computer system is part of the whole system often including other hardware and mechanical parts.

**Monodevelop**

Cross-platform mono development IDE.

**NAT**

Network Address Translation. Used usually to share one public IP address with many computers in local network.

**SCP**

Secure copy. Used to transfer files between computers using SSH protocol.

**SSH**

Secure Shell. A secure network protocol that is used for example remote command-line access.

# 1 INTRODUCTION

## 1.1 Objectives of this thesis

This thesis was created for the N4S@JAMK project. The project was part of the Need4Speed program managed by DIGILE and financed by Tekes. The program researches tools and business models for companies in order to make the companies faster and more reactive for the changing needs of customers and digital economy.

The objective of this thesis was to create a continuous deployment chain for Lego Mindstorms EV3 product. Continuous deployment means that the deployment process of software is automated as much as possible and deployment is done continuously. A new version of software is deployed to production every time when new features or bug fixes are introduced to the software. This means that customers do not have to wait for a big release when a set of features is released at once but features are introduced to customers as soon as they are completed.

The Lego Mindstorms kit that the continuous deployment chain is implemented for is a programmable Lego set. It contains a programmable hardware and parts to make programmable robots or other systems that have some fascinating functionality. The kit contains motors, sensors and a programmable computer that runs Linux operating system. Although the product is intended for kids, it is a great example of an embedded system. Nowadays, when embedded systems are everywhere, it is hard to imagine life without them. Every device that has some kind of computer or a computer chip controlling them can be considered to be an embedded system. There must be a huge amount of companies developing software for all these systems. Making easier, faster and leaner software development for them could be a valuable asset for those companies. Most likely some of these companies are still using manual labor in processes that could be automated. This thesis researches how continuous deployment could be adapted for the embedded systems development. Different tools are studied that could be used in continuous deployment chain and



different practices are discussed that could be adopted for software development of embedded systems.

## 1.2 N4S@JAMK project

N4S@JAMK is a project at JAMK University of Applied Sciences. The project started in 2014. It is part of the Need for Speed research program ran by DIGILE. The employees of the N4S@JAMK project are mostly students of JAMK University of Applied Sciences or fresh graduates. JAMK University of Applied Sciences has participated in DIGILE programs also in previous years; in 2010-2013 JAMK participated in Cloud Software program. In that program the main goal for JAMK was to develop the FreeNEST product platform. The author of this thesis also took part in the development of FreeNEST product when that project was running.

The DIGILE Need for Speed program has three work packages that target different problems in digital economy. One of the work packages is Mercury Business which focuses on how a business could behave like liquid mercury. This means the ability to adapt to new business conditions and search new business opportunities and react to them with minimum effort. (Mercury Business. N.d.)

The second work package is Deep Customer Insight. It focuses on collecting usage and behavioral data and feedback from the customers. This way the company can quickly react to customer needs. The package also focuses on analysis and visualization of the catered data and tries to understand what data should be collected. (Deep Customer Insight. N.d.)

The third work package is Delivering Value in Real Time. This work package focuses on providing approaches, methods and tools for quicker designing, creating and prototyping. One of the main goals of this work package is to find ways to increase the delivery and deployment speed. This thesis focuses on finding ways to make embedded software development quicker by implementing a continuous deployment chain for an embedded system, therefore this thesis is closest to target of this work package. (Paradigm Change – Delivering Value in Real Time. N.d.)

## 2 CONTINUOUS DEPLOYMENT

### 2.1 Definition of continuous deployment

Continuous deployment is all about automation. When a developer submits changes to the version control system a series of automated events start to happen. The end result is that the changes that the developer made are automatically released to production. Before more in depth explanation the next pages go through three different, however, very similar buzzwords.

#### **Continuous integration**

Before an explanation about continuous deployment, it is good to know what continuous integration means. All of the most widely used version control software have some kind of branching system. When a developer has a request to add some new feature to the main program, s/he then usually makes a new branch of the main development branch also known as mainline. In this branch the developer can then develop the new feature freely without having to worry that someone else has made changes to the same files as s/he. When the feature is ready and set for the integration back to the mainline, the developer would merge the changes to the mainline locally. Then the developer would locally test the program and fix all the found bugs. After making sure that nobody else had time to make changes to the mainline the developer can submit his changes to the main repository. This is a practice usually used when working with version control software.

This practice can still be somewhat lacking. Although the software works in the developer's computer, it might be that it will not work on the production machine. Maybe the developer forgot to commit something or the settings in development machine were different. Anyway, the mainline branch should always have a working version there so it can be quickly deployed.

Continuous integration is a practice that can make the process easier. It tries to solve the problem with automation and practices. When a CI server detects change in

mainline it starts to build the software automatically. After that, the server then proceeds to test the software automatically. If any errors are found, those errors are reported back to the developer. Then the developer can fix the bugs and try again. (Chletsos, M. 2012)

This procedure will catch issues early. To get the greatest benefit of continuous integration the developers have to devote to certain practices. Committing often is the key to avoiding merge hells and making sure that one developer's changes work well with other developer's changes. This also makes sure that the continuous integration chain is often used.

Testing in continuous integration case is mostly carried out with unit tests. A unit test tests the software on code level. Every class method or function should have its own tests written. This of course requires devotion from the development team. When done right continuous integration can have a very positive effect on code quality. Also, when the issues are caught early some time and money can be saved afterwards. (Chletsos, M. 2012)

### **Continuous delivery**

Continuous delivery takes the continuous integration a step further. The continuous integration chain can only run unit tests. The continuous delivery model takes on where the continuous integration left. After the same steps that CI ran the CD model takes the code that passed unit tests and deploys it to a staging environment.

In the staging environment the testing can continue. A set of automated acceptance tests can now be executed against the software. The acceptance-testing phase can for example test the performance or reliability of the software. For example, in the case of web software the testing could include automatically clicking through navigation bar and expecting certain pages to pop up.

With this kind of setup one can always be sure that the code in the repository is deployable to production or testing environments. In the continuous delivery case the deployment to production environment is usually automated and behind one

button press; however, the deployment has to be triggered manually. This can be useful in cases when some new feature is developed. For example, multiple versions of the software can be deployed quite easily for A/B testing, or a version under development can be deployed to staging environment for user acceptance testing. (Chletsos, M. 2012)

### **Continuous deployment**

Continuous deployment is the final state of continuous automation. It takes the continuous delivery even further and also automates the final deployment part. If all the tests pass, the code is deployed to production automatically. This means that there can be multiple production deployments in a day. In continuous delivery developers can choose if they deploy the code or not; however, in continuous deployment this is automated and every build that passes all testing is deployed. (Fowler, M. 2013.)

With this amount of automation there must be a great deal of testing involved, there cannot be any faulty code passing through the chain straight to the production environment and for the customers to use. This puts some pressure on the developers. Every single class should be fully tested to avoid faulty code. In addition to this, it might be a good idea to test if the code is actually ready to be deployed to production. It might be that some bug has passed the unit testing phase and the program does not even start. Executing some sort of acceptance testing might be a good idea. Because everything else in continuous deployment is automated, the acceptance testing should be automated too. Unfortunately acceptance testing is not as easy to automate as the unit testing. In the case of web development the tools are already there. A tester can put a script clicking through web page and confirm that right pages appear. In case of embedded systems the acceptance testing might not be so straightforward and testers might have to start to be creative to test everything that is wanted. This can of course be a project-specific problem. Some projects might be easier to test than others.

## 2.2 Continuous deployment pipeline

The actual implementation of continuous deployment pipeline is a case-by-case matter. Different projects require a different set of tools. For example the unit testing frameworks are language specific. Nevertheless, all the implementations should have about the same parts.

### **Version control software**

Some type of version control software is a basic tool in any kind of software development. The version control software allows developers to work with same files at the same time and not worry about each other's changes. The changes are merged back to the mainline occasionally and they are tested in order to see that everything works with other developers' work.

There are multiple different version control systems available. Four of the most popular are: CVS, Mercurial, Subversion and Git. CVS and Subversion use a Client-server model and Mercurial and Git use a distributed model for repository control, which means that in client-server model the user makes commits to server and in the distributed model the user makes commits locally and occasionally submits the changes to the server for others to see. (McNab, S. 2014)

In continuous deployment pipeline the version control software has the role of holding the software source code. Also, the deployment scripts for the project can be stored in version control repository, thus they are easy to edit. The version control software also holds the version history of the software. This allows developers to easily go back to an older version if necessary.

There are many different workflows with version control software. Usually the workflow practice depends on team size and company policies. Usually the practice includes some kind of branching system where developers develop some feature in a branch taken from the mainline. After the feature is complete the branch is then integrated back to the mainline. In some cases there might be some forking practice involved where the developer forks the main repository, which means that the

repository is basically copied for the developer. The developer makes the feature in the fork and then makes a pull request to merge the two repositories back together. (Distributed Git - Distributed Workflows. N.d.)

In the case of continuous deployment the workflow does not matter that much. The pipeline is usually made in such a way that the deployment pipeline is triggered when the mainline on the server gets a new commit. As mentioned in previous chapter, developers should often commit to this mainline. The features that the developers make should be so small that they can be implemented in a day or less time. This means that the developers can often integrate the changes to the mainline. This kind of practice saves the developers from merge hells and it means that the continuous deployment pipeline is often used. This can reveal issues early and save some company money.

### **Continuous integration software**

The continuous integration software is the part in the continuous deployment pipeline that works tightly together with version control. Depending on implementation of the Continuous automation pipeline, the continuous integration software can poll changes in version control or the version control software can trigger a build job on the continuous integration software.

CI software is usually not that complex. When a CI server gets triggered it runs a predefined job. In this job the user can define what should be run and where. The master computer can run the job locally, or in some CI software the master can have slaves where it distributes the workload.

The features and how the software works can be different across the different CI software. Both, commercial CI software and open source and free to use CI software exist. Some of the CI software have a web based UI and some are controlled and configured with other means.

The CI software is usually installed on a server. In most cases it does not matter where this server is located physically as long as it has an access to the version

control repository. Nowadays, the cloud services are so cheap that a virtual machine can be purchased from a cloud, therefore in some cases there is no need for physical servers anymore.

### **Build machine**

The build machine, also known as build server or build agent is the computer that runs the job in continuous deployment pipeline. The build machine can be the same computer as where the continuous integration software runs, or a slave machine, which the CI server controls. The CI software tells the build machine what commands should be executed. In many of the CI software you can either use some build automation tool for building process, or run a command on command line.

### **Unit testing**

Depending on the project, the unit tests can be executed on the build machine or somewhere else. In case of embedded systems project, it might be that the unit testing requires specific hardware. In that case the build machine could command some other device. It is recommended though that the hardware dependencies are abstracted away in the earliest possible step. That way no specific hardware is needed for unit testing, which might save time and money.

There are many unit-testing frameworks available for many different languages. Many of the frameworks are based on xUnit framework. The unit tests test software on function and method level. More on unit testing can be found later, in chapter 4.2 Unit testing.

When unit tests are executed, the result file that a unit-testing framework produced is reported back to the developer. As mentioned before, most of the CI software have some kind of web UI. There developers can see the build history and the test execution history. Some of the CI software can produce charts of the test execution history.

### **Staging environment**

Staging environment is the environment that is as similar as possible to production environment. Before the software is deployed to production it should be tested in a production-like environment where automatic acceptance tests are executed. In case of embedded systems, the staging environment could simply be a device, which the software is developed for. (What is a staging environment? N.d.)

In case of embedded systems the acceptance testing can be very hard or sometimes even impossible to implement. Implementing automatic acceptance tests on embedded systems would probably require specific hardware specifically made for testing purposes. In that case, this device would be the staging environment. When doing web development, a staging environment would be just a server with the same software installed and with the same amount of resources as the production server has.

### **Deployment to production**

The final part in continuous deployment pipeline is to deploy the software to a production environment. This could need a very project-specific solution, on how it is implemented. For example, in case of simple web development, this step would only include the deployment of web page to a server hosting the website. In some cases this would only need a simple script that is executed when the software is deployed.

In case of embedded systems the deployment could be implemented, for example with some kind of packaging system and package server. When a new version of software is deployed, it would be copied to a package server. Then the devices using the software would update themselves from the package server. This could be one solution to deploy new software. In the thesis assignment a more simple solution was used. The new version of software was simply copied over SSH using fabric framework.



## 2.3 Why continuous deployment

In software development, continuous deployment has some clear benefits against a model where features are deployed in large releases. When a set of features is developed at the same time in some timeframe and then deployed to production at the same time the end of the development cycle can be a hustle. Rarely the schedules are perfectly accurate and there probably is always hurry to get all the features ready before the deadline, which can mean that developers have to choose between quality and time. (Ries, E. 2009)

Another benefit of continuous deployment is that in case of longer release times, often features that have been completed early, would have to wait for a release. Also, bug fixes might not be deployed at once and some features might be broken in production for a long time. It also might be that the time needed to implement some important feature was estimated incorrectly and there was no enough time to finish that feature. The deployment would have to wait for the next deployment window or it might be deployed to production without all the functionality as planned. With continuous deployment, the feature is released when it is ready and the deployment does not have to depend on other releases. (Neely, S. 2013)

Another benefit with continuous deployment could be that in a large release cycle style it can take a long time to get a feedback from the users. It could be that developers made a feature ready, however, they were waiting for other features to be completed before deployment to production. It could take a long time for a customer to see and test the feature and give feedback. With continuous deployment, where all new features are instantly deployed to production, the development team can be very quick at reacting to the customer feedback.

## 3 SANDERO PRODUCTION ENVIRONMENT

### 3.1 Sandero

Sandero is a software production environment developed in N4S@JAMK project. The platform loosely integrates different widely used CI and software development products for an easy installation and use. In the thesis assignment, the Sandero production environment was used in the implementation of continuous deployment chain. At the time of making this thesis project, the Sandero environment was also in development phase. The author of this thesis also took part in the development of the Sandero production environment.

The core of Sandero product consists of GitLab, Jenkins and OpenLDAP. The following figure is a poster showing how the Sandero production environment was used in N4S@JAMK project. In addition to GitLab, Jenkins and OpenLDAP the project used few commercial software for project management. Flowdock was used for communication between project members, Trello was used for tasking and Dropbox was used in cases that files had to be shared between the employees. Also pictures and such could be uploaded to Dropbox where they can be embedded to web sites with ease. The robot in the picture is the icon of Hubot. Hubot is an IRC bot that can be used with Flowdock. The other robot in the poster is Robot Framework. Robot Framework is a generic testing framework that can be used in acceptance testing and acceptance test-driven development. (Robot Framework – Introduction. N.d.)

## Reference Production Environment - SANDERO v1.1

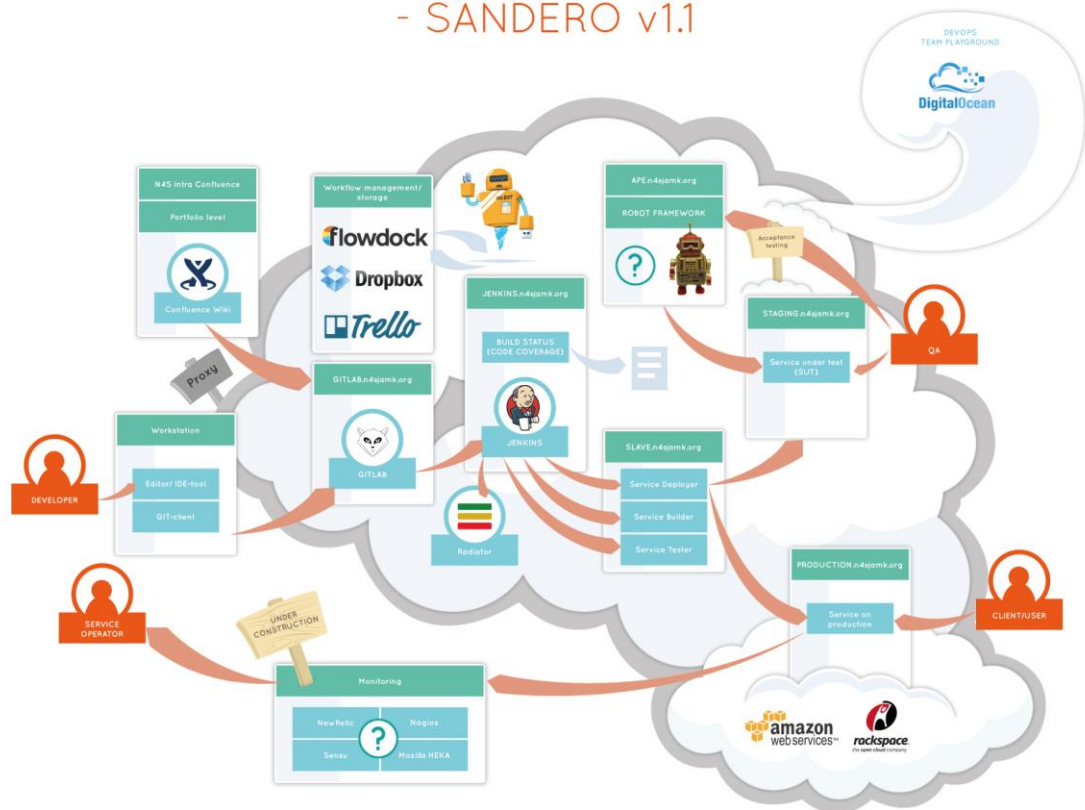


FIGURE 1. Sandero environment

The Sandero product is mainly designed to be deployed in a cloud environment. At the time of working with this thesis assignment, the Sandero environment was hosted in DigitalOcean cloud and the Sandero environment was still under development. All the features were not implemented yet. The deployment of the environment should be automated, however, the feature was not ready at the time. In the next chapters there is more information on different tools integrated in Sandero.

### 3.2 OpenLDAP

OpenLDAP is an open source implementation of the Lightweight Directory Access Protocol or LDAP. LDAP can be used in many purposes. It can store data, for example contact information or any kind of directory-like information. Still most often, LDAP is

used for providing single sign on where a single password is shared between many services. In Sandero environment LDAP is used just for that. With this kind of setup it is possible to manage users in one place. Both GitLab and Jenkins support LDAP authentication, although with Jenkins a plugin is required. (What is LDAP? N.d.)

OpenLDAP does not provide any kind of graphical interface for the user management. It only includes a stand-alone LDAP daemon server called slapd, libraries and some other tools. OpenLDAP can be administered from command line; however, this is difficult. In the N4S@JAMK project a web application called phpLDAPadmin was used for user management. (OpenLDAP. N.d.)

### 3.3 Git

Git is a free and open source version control system. Git uses a distributed model for controlling the source code. In Git model the developer has a full clone of the entire repository in their local computer, which allows the developer to work without connection to a shared repository. In Git, branching and other commands are fast because they are executed locally. In Sandero environment Git comes with GitLab.

Git was primarily developed for Linux; however, now it supports all the major operating systems. Git is mostly used with command line tool, although official Git client comes with a build in GUI tools included for those who prefer to use them. Also, third party implementations of Git are available. (Git – About. N.d.)

With Git and because of its distributed model of repository control, it is possible to use different kinds of workflows. For example GitHub uses a workflow where a user can clone someone's public repository to his own independent repository. This is called forking. The user who forked the repository can then develop something in his own repository and then make a request for the original user to merge the repositories back together. This merging request is called pull request. Users can discuss the pull request and the changes made with each other. Anyone can participate in the conversation. The owner of the repository can then accept or reject

the pull request. This is a great workflow for open source project where there are plenty of participators. (Git – About. N.d.)

Git does not come with any kind of server software or it cannot manage what users have access to repos. These requirements have to be satisfied with other tools. To make a shared repository kind of workflow, like Subversion has, the shared repository has to be served some way. The easiest way is to use SSH to access the shared repository. SSH access requires only SSH server to be installed on the server computer where the shared repository is. The repository access would be handled with operating system when SSH access would require password or SSH key. The repository can be shared over HTTP or HTTPS too, however, configuring a system like that can be time consuming. The better way is to install some software that has Git integrated and can serve Git repos over SSH or HTTP or HTTPS out of the box. One option is to install GitLab. It has these features and even more. There is more on GitLab in the following chapter.

According to Eclipse Foundation, Git is the most used version control system surpassing subversion in 2014. Third of the developers report that they use Git as their primary version control system. According to Ohloh, a website that maps the landscape of open source software development, git is the second most used version control software in open source projects claiming 37% of all repositories. The most used version control software in open source projects is still Subversion with 48% percentage. (Eclipse Community Survey 2014 Results. 2014, Ohloh - repository compare. 2014)

### 3.4 GitLab

GitLab is open source software that offers an easy Git repository management from a web UI. GitLab also features project access management, wiki for projects and issue tracing capability. GitLab comes with two editions. The Community Edition is free to use; however, it lacks some features that Enterprise Edition has. The Enterprise Edition is not free and it has a subscription based payment system. Figure 2

illustrates the GitLab web UI. From there users can create new repositories and manage access to them.

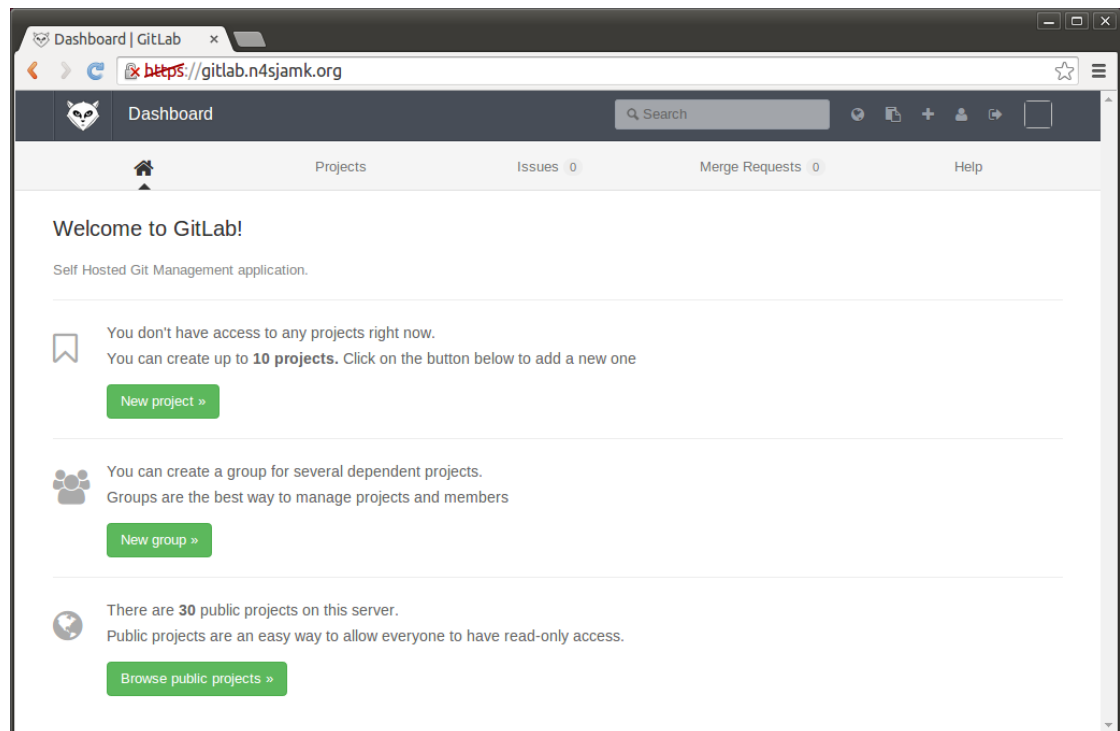


FIGURE 2. Gitlab web UI

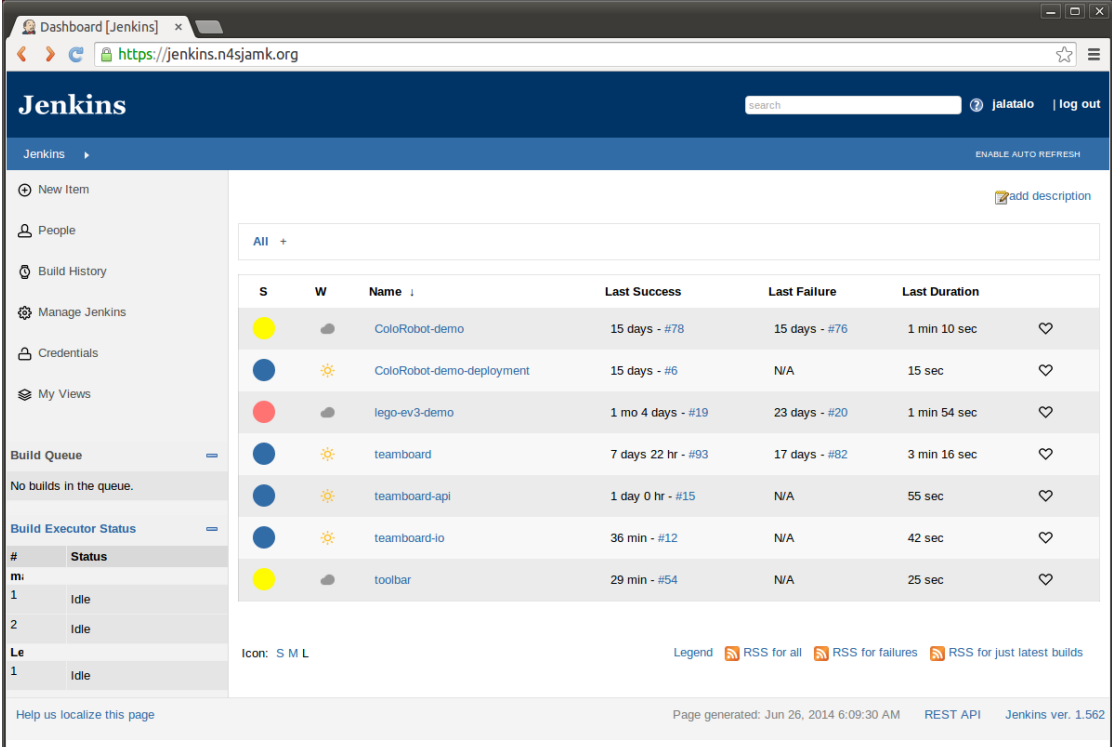
Gitlab was chosen in Sandero production environment because of its easy integration with LDAP and for its wide set of features for light project management. It includes an issue tracer and a wiki for documentation. With GitLab it is easy to make projects and restrict which developers have access to these projects. A different level of access modes can be given to different users for projects. GitLab supports different kinds of workflows. Forking repositories is supported and users can make merge requests.

GitLab automatically includes Git in its installation. It supports repository cloning over SSH or HTTP or HTTPS. When cloning over SSH, developer can save an authentication key to GitLab. After that, using Git is easy when no passwords are required when using git commands. GitLab also allows easily to set up web hooks. Web hooks allow developers to trigger an http POST request to some predefined URL when code is pushed to the repository or a new issue is created. In the thesis

assignment, web hooks were used to start a Jenkins job when new code was pushed to the Git repository.

### 3.5 Jenkins

Jenkins is a Continuous Integration software used in continuous automation. Jenkins automates execution and monitoring of jobs that are executed repeated times. It is mostly used at building and testing software projects automatically. It has an easy to use web UI, where new jobs can be created and configured. Jenkins supports third party plugins, which can extend the features, and tool support. Figure 3 illustrates the Jenkins web UI. From there users can create and manage Jenkins jobs and see the job execution history.



The screenshot shows the Jenkins web UI dashboard. The main content area displays a table of jobs with columns for status, name, last success, last failure, and last duration. The jobs listed are:

S	W	Name	Last Success	Last Failure	Last Duration
Yellow	Grey	ColoRobot-demo	15 days - #78	15 days - #76	1 min 10 sec
Blue	Yellow	ColoRobot-demo-deployment	15 days - #6	N/A	15 sec
Red	Grey	lego-ev3-demo	1 mo 4 days - #19	23 days - #20	1 min 54 sec
Blue	Yellow	teamboard	7 days 22 hr - #93	17 days - #82	3 min 16 sec
Blue	Yellow	teamboard-api	1 day 0 hr - #15	N/A	55 sec
Blue	Yellow	teamboard-fo	36 min - #12	N/A	42 sec
Yellow	Grey	toolbar	29 min - #54	N/A	25 sec

The dashboard also includes a sidebar with navigation options like 'New Item', 'People', 'Build History', and 'Manage Jenkins'. At the bottom, there is a footer with page generation information: 'Page generated: Jun 26, 2014 6:09:30 AM' and 'Jenkins ver. 1.562'.

FIGURE 3. Jenkins Web UI with Simple Theme Plugin installed and using jenkins-clean-theme

#### Jenkins jobs

Jenkins jobs are used to define and configure how a project is build and, or tested. One software project can have more than one Jenkins job. One job can trigger

another job based on certain conditions, which makes it possible to separate software testing and deployment to different jobs. Users can then add a post-build action to testing job and trigger the deployment job only if the build is stable.

Jenkins can automatically build Apache Ant and Apache Maven based projects, however, it can also execute shell or windows batch commands. In the thesis assignment, building and testing scripts were created using a Python framework called Fabric. The Fabric scripts are executed by using a command line tool `fab`. The fabric command was executed in Jenkins job using execute shell build step. (Jenkins - Building a software project. N.d.)

The Jenkins software monitors every job it runs. Users can see the console output from web UI captured during a job execution. Jenkins also marks every job to Success, Unstable or Failed based on if the job build or testing process passed failed or partly passed. When Jenkins is running shell script build jobs, it decides if the job was failure or success based on the return value of the shell command. In testing, when using xUnit based unit testing frameworks and a xUnit plugin a user can define thresholds to Success, Unstable and Failed flags. For example, user can define that if any of the tests failed the build is Unstable and if fewer than 90% of tests failed the build is Failed as well.

Starting a job can be done manually from the Jenkins web UI or it can be automated in various ways. The job can be configured to be executed between certain intervals, for example every night, or it can be built every time some other job is built, or another job can trigger the building of this job. The build can also be triggered remotely or Jenkins can be configured to poll version control repository for changes.

### **Jenkins agents**

Jenkins allows users to distribute the building process to other computers using Jenkins agents. Jenkins agents can be used to take the workload off from the master computer or they can be used in a testing process when testing has to be done on multiple different systems and operating systems. (Jenkins - Distributed builds. N.d.)



Jenkins has a few different ways to launch a slave agent. The easiest way on UNIX systems is to start Jenkins slaves via SSH. This requires user to make SSH key pairs and make sure that the master computer has access to the agent computer over SSH. When the connection is established the Jenkins master copies the necessary binaries for slave machine and handles the rest. This is a very easy way to start a Jenkins agent but it does not work every time. The master computer has to have an access to the slave machine over SSH. If these two computers are not in the same network and the agent machine does not have a public IP address, then the master cannot easily access the agent over SSH.

If the agent computer is not easily accessible over SSH the connection has to be initiated from the agent's side. This can be done by starting the slave daemon at the agent computer manually or automatically at every startup. The agent daemon is a java jar package that is started with right parameters to connect to the Jenkins master. The jar package can be loaded from the Jenkins instance using the right URL. In the thesis assignment this way was used in starting the Jenkins agent. In the demo the deployment computer had to be in the same local network as the EV3 brick was. In that case the Jenkins master machine was in DigitalOcean cloud environment so the SSH method was not possible. When the connection between the agent and master machine is initiated from agent's side the connection works even through NAT.

### **Jenkins plugins**

Jenkins allows users to install third party plugins. These plugins can change the look and feel or the functionality of the web UI or they can add support for different developing tools that Jenkins does not support out of the box. For example Jenkins does not support Git by default, but with Git Plugin Jenkins can do everything with Git as it can do with Subversion, which it supported by default. Table 1 illustrates some of the plugins installed in Jenkins instance used in the thesis assignment.

TABLE 1. Jenkins plugins used in the thesis assignment

Jenkins Plugin	Explanation
<b>Build Authorization Token Root Plugin</b>	Allows to start build job even if the overall read access is denied from anonymous user
<b>embeddable-build-status</b>	Allows Jenkins to expose a build status of a job to external websites easily by using a simple html tag
<b>Git Plugin</b>	Makes Jenkins to support Git version control software
<b>LDAP Plugin</b>	Makes Jenkins to support LDAP authentication
<b>Matrix Authorization Strategy Plugin</b>	Allows more specific authorization rules than normally supported
<b>Matrix Project Plugin</b>	Allows the authorization rules to individual projects
<b>Simple Theme Plugin</b>	Allows users to use different kind of styling on Jenkins
<b>SSH Credentials Plugin</b>	Allows users to save SSH credentials to Jenkins
<b>xUnit Plugin</b>	Makes Jenkins to support different xUnit based unit testing frameworks

### 3.6 Proxy

Because of security concerns it was decided that all connections to Jenkins, GitLab and OpenLDAP servers would go through a proxy machine. The author of this thesis did not take part in the installation and setup of the proxy machine. The installation and configuration was done by other employee in the project and the author of this thesis only took part in debugging and planning how the proxy could work.

The proxy software used in the reverse proxy implementation was Apache. The Apache instance was using `mod_proxy` plugin so it could proxy the passing traffic. The proxy used subdomains as a way of figuring out where the traffic should be directed. For example, HTTP packages to `jenkins.n4sjamk.org` would be directed to

Jenkins server and packages to gitlab.n4sjamk.org would be directed to GitLab server.

Unfortunately, for technical reasons the Apache proxy could proxy only HTTP and HTTPS packages. This developed a problem when trying to connect a Jenkins agent to Jenkins server. The packages could not be directed to right servers because the protocol used was no HTTP or HTTPS. The problem was fixed with iptables port forwarding trick that may not have been the best solution but it worked.

### 3.7 Fabric

Fabric is a Python framework used for deployment automation and other administration tasks. Fabric provides functions to execute local and remote shell commands and upload and download files. Fabric comes with a command line tool which is used to execute the fabric scripts. Here is an example of fabric script:

```
01 from fabric.api import local, run, put
02
03 def run_whoami_locally():
04     local("whoami")
05
06 def run_whoami_remotely():
07     run("whoami")
08
09 def transfer_file(
10     local_file = "local_dir/file",
11     destination = "remote_dir/file"):
12     put(local_file, destination)
```

The script would run when executing the `fab` command in same directory as the script with right parameters. The file should be named `fabfile.py` so that the `fab` tool automatically knows to look for it. If the file is not named that way the filename can be passed as an argument to `fab` command with `-f` or `-fabfile=` options. (Welcome to Fabric! N.d.)

The previous fabric script defines three tasks, `run_whoami_locally`, `run_whoami_remotely` and `transfer_file`. The tasks are executed by giving the name of the task as an argument to the `fab` command. For example, to execute the `run_whoami_locally` task, the `fab` command would look like this:

`fab run_whoami_locally`. This task simply executes the UNIX `whoami` command on the command line in the local computer.

The two other tasks are different. They execute commands on remote hosts. To execute these tasks the `fab` tool accepts a host parameter, or if not given the tool asks the user to provide the remote hosts during an execution. Host or hosts are provided for `fab` tool using `-H` option. Commands `fab -H user@192.51.100.1 run_whoami_remotely` and `fab -H user@192.51.100.1 transfer_file` execute the two other tasks. The `run_whoami_remotely` executes UNIX `whoami` command in the remote host and the `transfer_file` task transfers a file to a remote host over SSH.

The fabric command line tool can provide arguments to tasks. In the example fabric script the third task, `transfer_file` takes two parameters, `local_file` and `destination`. A user can provide these arguments to the task by adding them to the command. Example command: `fab -H exampleuser@192.51.100.1 transfer_file:local_file="example.txt",destination="~/example.txt"`. The previous command would have transferred `example.txt` file to remote host's example user's home directory.

## 4 TEST AUTOMATION

### 4.1 Test automation with embedded systems

Testing is a very important part in continuous deployment. When everything in deployment process is automated, it is possible that faulty code gets all the way to the production environment by accident. That is why there are multiple states of testing in continuous deployment chain. Because everything else in this chain is automated, also the tests have to be automated.

Unit testing is easy to automate and there are existing frameworks probably for every language. Unit testing with embedded systems is not so much different to

some other kind of unit testing if testing is taken into account in the planning phase. Embedded systems can have all kinds of input devices, for example sensors that can make these systems difficult to test. The software architecture can be built in such a way that it makes testing easier. When software architecture is made in a way that these sensors or other devices are easy to fake on a software level, it is easy to make unit testing possible for these systems.

Faking a software module, or parts of software is called mocking. The software architecture should be designed in such a way that everywhere in an application a sensor or other device is accessed in a same way with a same function or class. The function or class should be possible to swap to other implementation of the same function or class for testing. This way the tester could provide an implementation of the function that gives predictable data to the rest of the software. This removes the dependency of hardware in testing process when for example sensors can be faked on a software level. This enables developers to run unit tests even on their computers when no special hardware is needed. More on mocking and unit testing in the following chapters.

Automatic acceptance testing is not always as easy to implement. The acceptance testing should verify that the software is ready for deployment to production environment. This can include performance testing and functional testing. With web development, tools for this kind of testing already exist. It is easy to verify that a website can serve for example ten thousand users by bombarding the server with bots and a simple script. Also, there are already frameworks for functional testing. For example with Selenium framework developers can automate web browsers to click through navigation bar and expect certain results. With this setup it would be easy to execute the tests with different operating systems and browsers.

With embedded systems the automatic acceptance testing process can be a little more complicated. How to test for example, that something that should happen if user presses a button, actually happens. Or how to test that sensor data is displayed correctly on a screen? Of course implementing the testing is project-specific. In some

cases the acceptance-testing phase can be automated and in some cases it could be impossible or too expensive to automate. In these cases some manual testing is always required, and a fully automated continuous deployment workflow might not be right solution in these projects.

To test the continuous deployment pipeline created in this thesis assignment a demo project was made to test it. The demo was demonstrated for other companies participating in Need4Speed program, in Q2 review event at Tampere. More on that demo later, however, in that Lego Demo automatic acceptance testing was not implemented perfectly. During the implementation of the demo, there was a plan that a second robot would have been built for testing purposes. That robot would have been slightly different and would have packed some extra sensors that could have been used for automatic acceptance testing purposes. Unfortunately, because of the schedule, there was neither time nor building blocks to make another robot. In the end the automatic acceptance testing was implemented just using the NUnitLite framework that was used in unit testing too. The tests were not perfect, however, they showed how the continuous deployment could be implemented.

## 4.2 Unit testing

Unit test tests the smallest piece of software that is testable. Usually this means testing functions and methods or classes in code. The main goal in unit testing is to isolate the unit under test from everything else. This way the tester can make predictable tests that test only the function or method and is not dependent on anything else. The best practices of unit testing includes that one unit test should test only one thing. One function or method can have, and should have multiple unit tests written to test it. (Microsoft Developer Network - Unit Testing. N.d.)

Unit testing should be easily automated. The tests should be runnable in developer's computer and they should run fast. Different unit test frameworks exist and there is at least one for almost every programming language. Several of the frameworks are part of the xUnit family that is a collective name for unit testing framework that are similar to Smalltalk's SUnit framework developed in 1998 by Kent Beck. (xUnit. N.d.)

In some cases, e.g. in test driven development, tests are written first and function after that. The tests define how the function works and when the tests pass the function is ready. This has some benefits. This way there is a written contract how the function should behave. Later if a test fails the developer can be somewhat sure that the error is in something he changed in code. When unit testing is done right refactoring code should be an easy and safe process. If all tests pass after refactoring a developer can be sure that the function works just like before. Unit tests also provide a documentation of code. When tests define how functions work a developer can read tests to see how the functions work. This way the tests also work like a documentation tool. More on test driven development in the following chapter (Unit testing. N.d.)

As mentioned in the previous chapter, testing should be taken into consideration already in software planning process. The software architecture should be planned in such a way that it is easy to write unit tests, which means that interfaces should be replaceable with mock objects so that functions and classes can be isolated from the rest of the code. Mock objects are a way to mimic class behavior in controlled way. There are different frameworks to make mock objects easily, however, it is possible to write them by hand. In Lego Demo mock objects were used to mock sensor data. This way the unit tests were runnable on a normal computer. In the demo project, a third party library was used in sensor data reading and it is not possible or practical to mock a whole library. In the demo an abstraction layer was created between the library and classes that were using the library to read sensor data. The figure for explanation is illustrated below. (Mock object. N.d.)

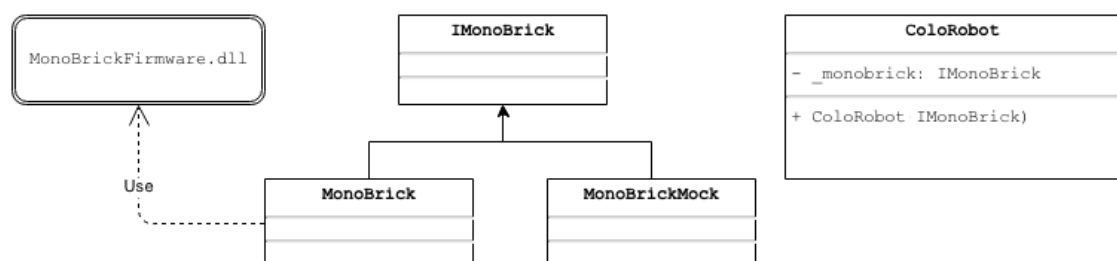


FIGURE 4. ColoRobot design pattern

In diagram, `ColoRobot` class has a property `_monobrick`. The property type is `IMonoBrick` which is interface class. This means that the property can be any object that implements the `IMonoBrick` interface. The object can be passed as an argument for `ColoRobot` class in the class constructor. In diagram there is two implementations of `IMonoBrick` interface, `MonoBrick` and `MonoBrickMock`. The `MonoBrick` class uses `MonoBrickFirmware` library and is always used in actual code. The `MonoBrickMock` is a mock class used only in testing. This way there is an abstraction layer between `ColoRobot` class and `MonoBrickFirmware` library that is mockable.

### 4.3 NUnit and moq

#### **NUnit**

In the thesis assignment an open source firmware called `MonoBrick` was used in the EV3 brick. The `MonoBrick` firmware executes programs on Mono framework, so the programs are written with .NET compatible languages. One of these languages is C#, and it was chosen for this project. C# has multiple different unit testing frameworks available, however, NUnit was chosen at first for this project. In the end half of the project the unit-testing framework was changed from NUnit to NUnitLite which is just a lightweight version of NUnit, more suitable for embedded systems. All the already written tests were compatible with NUnitLite framework, because the NUnitLite has the same syntax and the same basic features. (`MonoBrick EV3 Firmware`. N.d., `NUnitLite`. N.d.)

The difference between NUnit and NUnitLite is that NUnitLite uses only a minimal amount of resources. NUnit comes with a console runner and with a GUI based test runner. At first the console runner was used for test-execution on the Lego EV3 brick, however, because the execution time this way was way over a minute another solution had to be found. Normally it might not matter that the execution time on the target platform is this long. Usually the unit tests are executed on a normal computer anyway. In this case the unit-testing framework was used in acceptance testing as well, therefore the tests had to be executed on the target machine. Also,



because the demo was to be presented at the demo bazaar to an audience, all the excess waiting time would have to be reduced to minimal.

NUnitLite embeds the test runner to the code. When using NUnitLite the test execution time was reduced to about a half from before. The NUnitLite project used to have its own independent project, however, with the release version 1.0 NUnitLite project is part of the NUnit 3.0 project. At the time of writing this, the NUnit 3.0 project was still under development. In the thesis assignment the NUnit 3.0 was used by compiling the code from GitHub.

### **moq**

Moq is a C# framework for mocking. It supports both, mocking classes or interfaces. It takes full advantage of .NET Linq and lambda expressions. C# has multiple different mocking frameworks available. Moq was chosen for this project for its easy use, easy installation and support for mono. It is possible to write mock objects by hand, however, it is easier and quicker to use some mocking framework. (moq. N.d.)

Although mocking is a great tool for testing, it is possible to over use it or use it incorrectly. Some mocking frameworks, including moq, have a feature that lets the developer see how many times the class that is tested calls the methods of the mocked object. This is a great feature when used correctly, however, in some cases it can be used in a way that the tests start to test more of the functions internal implementation, and not the external behavior as they should. This can be a problem for example in case where the code is refactored. Although the function would still obey the contract of previous implementation the tests could still fail. (Mock object. N.d.)

### **Example of using NUnitLite and moq**

Here is a simplified example how to use NUnit and moq from the ColoRobot project. The example runs two tests against LoadCard method in ColoRobot class. The LoadCard method loads a new card in if no card is already loaded. The two tests test

that the feedmotor is started if no card is already loaded, and not started if the card is already loaded.

```

01 public class ColoRobot {
02     //... some code
03     private IMonoBrick _monobrick;
04     //... some code
05     public ColoRobot(IMonoBrick monobrick) {
06         _monobrick = monobrick;
07         // ... some code
08     }
09     public LoadCard() {
10         // ... some code
11         if(!_monobrick.IsCardIn()) {
12             _monobrick.FeedMotorOn();
13             // ... some code
14         }
15         // ... some code
16     }
17 }
18
19 [TestFixture]
20 public class UnitTests {
21     [Test]
22     public void LoadCard_NoCardIn_StartMotor() {
23         var mock = new Mock<IMonoBrick>();
24         mock.Setup(x => x.IsCardIn()).Returns(False);
25         var colorobot = new ColoRobot(mock.Object);
26         colorobot.LoadCard();
27         mock.Verify(x => x.StartMotor(), Times.Once());
28     }
29     [Test]
30     public void LoadCard_CardIsIn_NoStartMotor() {
31         var mock = new Mock<IMonoBrick>();
32         mock.Setup(x => x.IsCardIn()).Returns(True);
33         var colorobot = new ColoRobot(mock.Object);
34         colorobot.LoadCard();
35         mock.Verify(x => x.StartMotor(), Times.Never());
36     }
37 }

```

In the previous example, there is a simplified skeleton of ColoRobot class on lines 1 to 17. In the class there is the previously described abstraction layer between MonoBrick library and ColoRobot class at line 3. The \_monobrick property is initialized in ColoRobot constructor on line 6. The simplified method of LoadCard is on lines 9 to 16. In the example the method behaves in a very simple way. If the IsCardIn method returns false the FeedMotorOn method is called, otherwise nothing happens.

In lines 19 to 37 there are two simple tests. The first one makes sure that if no card is loaded inside ColoRobot, the feed motor is started if the LoadCard method is called. The second one makes sure that feed motor is not started if a card is already in ColoRobot and the LoadCard method is called.

Both tests are very similar. The only difference is that other test setups mock object to return false when its IsCardIn method is called and other returns true. On the first line in both tests a mock object is created using IMonoBrick interface class. On the second line the mock object is configured to return a Boolean value when IsCardIn method is called. The third line constructs a CoboRobot object and gives the mock object as a parameter for it. The fourth line calls the ColoRobot's LoadCard method. The final line in both tests verifies that in case the first test, StartMotor method is called exactly once and in the second test, StartMotor method is not called at all.

#### 4.4 Test-driven development

Test-driven development is a software development technique where the developer writes the tests first and the actual implementation for function after that. When a test is first written, it must first fail. This is because the feature is not yet implemented. If the test does not fail, it means that the test is poorly written or the feature is already implemented. After the test is written and it fails, a minimal amount of code is added so that the test passes. The implementation at this point does not have to be perfect. When the test is passing it is time to refactor the code. All the duplication is removed and the code is cleaned up. After that the whole process is repeated for another feature. (Test-driven development. N.d.)

The test-driven development is claimed to have many benefits over more classical test-last software development approach. When using TDD model the developers have to pay more attention to use cases and user stories when they are writing tests to features to be able to understand the feature requirements and exception conditions. Also, TDD can lead to more modularized and that way more easily maintainable and expandable code. When using TDD more tests are written and that helps to catch defects in code. (Test-driven development. N.d.)

Test-driven development has some weaknesses too. Maintaining a large amount of tests can be time consuming and expensive if the tests are written poorly. When the same developers write both, the tests and the function that is tested, they can share the same blind spots. For example, if the developer does not realize that the input parameters should be checked, neither the code nor the test either will verify those parameters. (Test-driven development. N.d.)

Test-driven development practice goes well with continuous deployment chain. Tests written in test-driven development model are unit tests that are easily automated and are of course part of the continuous deployment workflow. In the Lego demo, test-driven development was not used in demo implementation phase. This was because the nature of this project was very experimental and the size of this project was very small.

#### 4.5 Automatic acceptance testing

Acceptance testing is a testing phase that should ensure that the software delivers the expectations that the customer has for the software. This means testing the software from the customer's point of view and testing the user stories. The automatic acceptance testing in continuous deployment chain also serves as a regression test suite that makes sure that no bugs that the unit tests did not catch, are introduced in code. Automatic acceptance testing should be executed in a production-like environment. In case of embedded systems it would be the system or device that the software is developed for. There is no one right way to implement the automatic acceptance testing phase. The implementation differs from project to project. (Humble, J. Farley, D. 2010.)

In Lego demo, the automatic acceptance phase was implemented using unit testing framework and it only tested performance and reliability in a very simple way. This was enough for demo purposes, however, in case of a real system maybe another system developed just for testing purposes would have been necessary.

## 5 LEGO MINDSTORMS EV3

### 5.1 Mindstorms EV3 brick

The Lego Mindstorms EV3 kit that was used in this thesis was a building kit made by Lego. The kit contains a programmable EV3 brick which is the computer part, a set of different sensors and building blocks used to build the robots. The kit has instructions to build five different robots, however, there are instructions to build twelve additional models with instructions that can be loaded from Lego website. The Lego EV3 is the third generation of the Lego Mindstorms robotics line. The previous version of the series was called Lego Mindstorms NXT 2.0. All the sensors, motors and building blocks from NXT 2.0 generation are compatible with the EV3. (Lego Mindstorms EV3. N.d.)

#### **EV3 programmable brick**

The EV3 brick is the computer part of the Lego set. The brick has an ARM9 processor, 64 MB of RAM, 16 MB of Flash and a microSDHC slot. It supports memory cards up to 32 GB. The brick also packs an LCD display, USB port and Bluetooth. The device can be equipped with WiFi by connecting a supported USB WiFi dongle via USB port. In the thesis assignment a Netgear N150 WiFi dongle was used and confirmed to work with the brick. Figure 5 illustrates the Lego Mindstorms EV3 brick. (Lego Mindstorms EV3. N.d.)



FIGURE 5. The Lego Mindstorms EV3 programmable brick

The device comes with Lego's own Linux based operating system installed. The brick can be programmed with Lego's simple icon based development software. This way of programming the device is lacking since programming the device this way does not have all the advantages as programming with a real programming language. Luckily Lego has open sourced the EV3 software for hackers to use. This way there are multiple different custom firmware available to users. In the thesis assignment a custom firmware called MonoBrick was used. There will be more on that in the next chapter.

### **Sensors and motors**

The Lego Mindstorms EV3 home edition kit comes with three different sensors and three motors. The sensors which are included in the kit are: touch sensor, color sensor and infrared sensor. Two of the motors are large and one is medium sized. The Lego brick device can have four sensors and four motors connected to it

simultaneously. The sensors and motors are connected with special cables with connectors very similar to RJ12. Unfortunately RJ12 connectors do not fit in the sensor and motor ports of the Lego brick device. Figure 6 illustrates the sensors and motors coming in the Lego EV3 kit.



FIGURE 6. Sensors and motors coming in the EV3 home set. From lower left to right: touch sensor, infrared sensor, medium motor and large motor. Up left is a remote control device and next to it a color sensor. On top of them, there is a cable that is used to connect the sensors and motors to EV3 unit

In addition to Lego's official sensor set some third party sensors for NXT and EV3 devices are available. Mindsensors.com website sells sensors for both of these devices. The website also sells console adapters for EV3 brick. With the adapter users can access the console interface of the operating system from the first sensor port. The console access cable can also be made by users themselves, using a USB to UART bridge dongle and sacrificing one sensor cable as described by Xander Soldaat in his blog. (Console Adapter for EV3. N.d., Soldaat, X. 2013)

### Building blocks

The Lego Mindstorms series uses the same building elements as the Lego Technic line. The EV3 kit contains large variation of different building blocks including different sized gears, tires, tracks, axels and many types of parts and fastener elements. It is very easy to build all kinds of different structures with the building blocks. The structure used in Lego demo was self-designed and no instructions was used in the building process. That is why this device was chosen in this project. It is very easy to build different kind of structures with ease and with not much planning. With motors these structures can have some fascinating functionality, which attracts the interest of by passers. Figure 7 illustrates some of the building bogs available in the EV3 home set.



FIGURE 7. Some of the building bogs available in the EV3 home set



## 5.2 MonoBrick firmware

As mentioned in the previous chapter a custom third party software was used in the thesis assignment. There are multiple different firmwares available for the EV3 device. One of these is MonoBrick firmware. The MonoBrick firmware differs from the standard preinstalled Lego's firmware in that MonoBrick has an integrated mono framework installed in the operating system. This allows the EV3 to execute software written with any .Net compatible language. This gives much more potential to the system when software can be written in a real programming language compared to Lego's own visual, icon-based programming language. (MonoBrick EV3 Firmware. N.d.)

### Installation

The MonoBrick firmware is installed in an external memory card. This is done by making the memory card bootable. The MonoBrick firmware is open source and its source code can be found from GitHub. The readymade image can be loaded from MonoBrick homepage. The firmware installation to the memory card requires the image loaded from MonoBrick website and a tool that can create a bootable memory card. Programs that can make bootable memory cards exist for every major operating system. The easiest way to make a bootable memory card is to use dd tool installed in every Linux distribution and in OSX system. When using Windows or when wanting to use a graphical interface in OSX or Linux systems, some third party software has to be used. It is easy to find guides on the internet by searching Raspberry Pi installation guides and applying them in EV3 installation.

To make a bootable MonoBrick memory card on OSX system, go to MonoBrick website and download the MonoBrick firmware image and extract it by running `gunzip imagefile.gz`. Insert a memory card to the computer. Execute `diskutil list`. Figure out what disk is the memory card. Execute `diskutil unmountDisk /dev/diskX` where `diskX` is the memory card. Then execute command `sudo dd bs=1m if=/path/to/image/file.img of=/dev/diskX` where `diskX` is again the memory card disk. The command can

take a while to execute and it does not provide a progression indicator. When the command is executed the memory card can be unmounted and inserted in the EV3 device. When the device is booted it loads the new firmware from the memory card.

### **Use**

The MonoBrick firmware is loaded only when the memory card, in which MonoBrick is installed, is inserted in the EV3 device. The standard Lego firmware is not overwritten and it can be reloaded by removing the memory card and starting the device again. The MonoBrick firmware has a very basic UI which user can use to control the device using Lego Brick's five buttons. With this interface user can connect to WiFi if a supported WiFi dongle is in use, execute programs stored in the memory card and see information about the device.

The MonoBrick firmware has a dropbear SSH server included in the installation. When using a WiFi dongle the console is accessible over SSH. This feature in the firmware had a crucial role in the implementation of the continuous deployment pipeline. It was used when transferring files from build machine to EV3 device with SCP. Fabric framework also uses SSH to execute commands on the remote system. The SSH console can also be used in configuration of the system and for debugging purposes.

As mentioned before the EV3 device has console access in the first sensor port. Users can access the Linux console by using a special console access adapter or making one by themselves. A console adapter was obtained for this thesis assignment and it was tested with MonoBrick firmware. Unfortunately, the console adapter does not work well with MonoBrick. It is probably because of the MonoBrick setting the first sensor port from console access configuration to sensor configuration during the firmware bootup. It was observed that if the booting procedure was interrupted during the boot-up the console was in some cases accessible. The plan was to test if the console access adapter could be used in some way in the implementation of the continuous deployment pipeline but when discovering that the MonoBrick had a SSH server included in the installation, it was unnecessary to use it.

## Development for MonoBrick

As mentioned before, the MonoBrick has a mono framework included in the installation. This means that the firmware can execute any software developed with .Net compatible language. The software for the Lego demo was developed on Ubuntu system. The software was written and compiled using MonoDevelop IDE. MonoDevelop is a cross platform IDE used in the development with mono and .Net. MonoDevelop is available for all the major operating systems. At least in the Ubuntu installation, console tools come with the software installation. These console tools were used in the automation process. In the building of the program a tool called mdtool was used. Fabric executed commands using mdtool in the build machine.

## 6 ASSIGNMENT

### 6.1 Goal

In this thesis the goal was to make a continuous deployment system that would deploy software to Lego Mindstorms EV3 device. The idea for this thesis came from Marko Rintamäki, one of the project leaders in N4S@JAMK project. Mr. Rintamäki has a long history in software testing and he teaches the subject at JAMK University of Applied Sciences. As mentioned before, JAMK has taken part in other programs that DIGILE has managed. In Cloud Software program a project at JAMK developed FreeNEST Portable Project Platform that integrated different open source project management tools in one place for easy use and installation. That project was also managed by Mr. Rintamäki and the author of this thesis took part in this project. In that project the development process was somewhat automated. An instance of FreeNEST product platform and some other open source automation tools were used in the further development of the FreeNEST product. As for now, the development of FreeNEST has ended.

At the start of the N4S@JAMK project it was decided that another software development environment would be developed. In that environment the same tools

used in the development of the FreeNEST product, would be integrated and the installation of these tools would be automated. The new environment would focus on automation and not so much on the project management as FreeNEST did. The environment was named Sandero Production Environment.

This thesis was completed for the purpose of showing how the Sandero Production Platform handles projects targeting embedded systems. Some of the companies, taking part in the Need4Speed program, are developing software to embedded systems. The plan was that the Lego demo would create interest from these companies when the demo would be presented in N4S Q2-2014 Review demo bazaar event. The use of Lego Mindstorms product made it possible to do an interesting demo without much planning and in a short time. The demo would still be interesting enough to make people stop and come to the presentation stand that JAMK had at the review event.

## 6.2 Starting point

### **Sandero**

Although continuous deployment and other continuous automation practices have been in use for a while in the software development industry, there is no one right solution how to implement one. The implementation depends on the project itself and what tools are available at the company. The workflow in a continuous deployment chain is always about the same, however, the tools used in the actual pipeline can differ. For example, there are many version control software that can be used to hold the source code and version history.

As mentioned before, the assignment of this thesis used Sandero Production Platform as the frame in the implementation of the continuous deployment chain. The author of this thesis took part in the making and building the Sandero platform with the other employees working in N4S@JAMK project. All the employees working in the N4S@JAMK project had participated in the development of FreeNEST product. It was decided that the Sandero Production Platform would use to a large extend the

same tools that were used in the development of FreeNEST. That way everyone working in the project was familiar with the tools.

In the N4S@JAMK project, Ubuntu was the operating system that was used for development and work. Ubuntu server was also used in server computers. It was important that the Sandero environment is deployable to a cloud environment. The first instance of Sandero was created to DigitalOcean cloud. DigitalOcean is a virtual private server provider, which means users can create virtual servers to a cloud environment. The service in DigitalOcean has an easy to use web interface where users can create new instances and it is cheap. At the time of writing this thesis DigitalOcean was the cheapest provider. Their most cost efficient virtual machine cost 5\$/mo. The cheapest virtual server had 1 CPU, 512MB memory and a 20GB SSD Disk. When creating a new instance, or droplet as they are called at DigitalOcean, users can choose a preinstalled operating system and some preinstalled software if they choose to. The new virtual machine is created under a minute and the password is mailed to user's e-mail.

As mentioned in previous chapters, Sandero consists of different open source software tools. The tools were all installed to different computers in the cloud environment. This way there was no need to worry about port conflicts and that the applications would somehow disturb each other. The applications included in Sandero are: GitLab for light project management and version control, Jenkins as a CI software and OpenLDAP as a single-signon provider. In addition to those applications, Sandero also has a proxy server where all the traffic to other applications is passed.

These were the tools that the continuous deployment chain would have to be built with. The major challenge to be solved was how the software could be deployed when the servers were in cloud but the target system was in a local network. Otherwise there were not much other problems. All the tools were familiar and because of the FreeNEST project a semi similar automation process was already built and the experience from that project could be applied to this project.

## Lego

As was mentioned before, the Lego Mindstorms EV3 brick is running a custom Linux distribution by default. The custom firmware runs programs made with Lego's own visual based programming language. The Lego's custom Linux is open sourced and the code is available at GitHub. This has made it possible for hackers to make custom firmware for the EV3. The visual based programming language is no match to a real programming language. After some research it was discovered that few custom firmware exist that allow users to run programs written with different programming languages.

Two of these custom firmware were tested. One was the MonoBrick firmware that was chosen to the thesis assignment, the other was called ev3dev. The ev3dev is based on Debian Wheezy and it can make use of the packages available from Embedded Debian repositories. The installation of the ev3dev firmware was successful, however, the firmware was not used in the thesis assignment. The firmware was interesting: it can run all the languages that have an ARM port available. The problem was that at the time of making the thesis assignment, the supported programming languages had no libraries yet to access the EV3 sensor and motor ports. Since then, contributors have written language wrappers for many popular programming languages. These are listed on the ev3dev homepage. (Debian on LEGO MINDSTORMS EV3! N.d.)

The ev3dev firmware would have probably been more interesting choice for the EV3. The ability to use any language would have been great. The MonoBrick firmware was chosen to the thesis assignment, not only because the lack of sensor support of ev3dev, but because at first the plan was to make a demo that interacts with computer running a Unity demo. Unity is a game creation system and it supports all the major operating systems. The Unity's scripting system is created in a way that it uses Mono. The MonoBrick firmware runs programs with Mono, thus the communication between Unity and the EV3 brick would have been easy to implement. (Unity (game engine). N.d.)

The demo idea in the Unity case was to create some sort of virtual 3D world that would run on a computer. Then the EV3 brick would communicate with the computer over WLAN. A spectator then could interact with EV3 brick's sensors and at the same time something would happen in the virtual 3D world. The plan was not implemented, however, MonoBrick was still chosen for the thesis assignment because of time spent researching if Unity demo would be possible gathered some knowledge about the firmware.

### **Fabric**

Fabric automation framework was chosen for this project because of its easy use. Fabric is a python framework. The fab scripts are written in python and the author of this thesis has some previous experience with this programming language. The other option for automation framework was Capistrano. Capistrano is written with Ruby and is a more complex but similar automation and deployment tool as Fabric. At the time the author of this thesis had no previous experience with this programming language. After finding the Fabric framework it was not a hard decision to start using it instead of Capistrano.

### **Mono development**

The author of this thesis had no previous experience of development with Mono framework. Mono is an open source and cross platform implementation of .NET framework developed by Microsoft. The author of this thesis had some previous experience on development with C# and .NET on Windows systems. This of course was a huge help in the development process because the development between the two does not differ much.

The major difference between developing with Mono on Linux versus .NET on Windows is development tools. On Windows the most used IDE is by far Visual Studio. Visual Studio is not cross-platform compatible. On other than Windows systems MonoDevelop is probably the best pick for an IDE. MonoDevelop was also chosen for the thesis assignment project. When installing MonoDevelop to Ubuntu

using package manager, it comes with some handy command line tools that are great for automation.

### 6.3 Implementation

The work on the thesis assignment started in the beginning of April. At first the assignment was a somewhat vague. As mentioned before the first assignment included the interaction between the EV3 brick and a computer running a 3D virtual world on Unity game engine. The Unity side was assigned to another employee working in the N4S@JAMK project and the author of this thesis worked with EV3 side. The work started on implementing a way of communication over network between EV3 brick and computer running Unity. Because both Unity and EV3 supported Mono and that way programs written with C# and .NET it was easy to make them to speak to each other. The Unity plan was still scrapped quickly and the assignment focused on the EV3.

In addition to carrying out a continuous deployment chain to EV3 brick, the other focus was on unit testing of the EV3 brick. The assignment was to research how it would be possible to run unit tests on a normal computer and somehow simulate the sensor and motor data. At first it was thought that this would need some sort of emulator to be implemented. Afterwards it was discovered that the problem could be solved more easily with good software architecture and mocking.

#### **Jenkins**

As mentioned before, the first problem to solve when starting to implement the continuous deployment chain was to figure out how Jenkins could connect to EV3 brick. The Jenkins instance was on the DigitalOcean cloud and it would have to deploy the software to EV3. This would need some sort of communication between the two. The problem was that the EV3 brick does not have a public IP address when it is connected to local network with WLAN. This is because normally when connecting any machine to some local network, it gets an IP address that is visible only in that network. Usually all these machines share one public IP and that IP



points to the router. This is called NAT. Jenkins cannot connect straight to EV3 if it is behind NAT. This could have been solved using port forwarding rules on the router, however, that would have been an ugly solution and it was not used to solve the problem.

The problem was solved using Jenkins agents as described in the chapter 5.5 Jenkins. The Jenkins agent would be deployed in the same local network as where the EV3 brick was connected. The agent would create a connection to Jenkins machine in the DigitalOcean cloud through NAT and that way the Jenkins server could execute commands on that machine. The only step to take in this setup was to get a java jar package loaded from the Jenkins server. This jar package would then be executed on the agent machine by hand, or on every boot with an init.d script. The executable needed Jenkins master URL and a secret token as arguments. The token authorizes the connection to the Jenkins master. At first when the project had no SSL Certificate for Jenkins machine a `noCertificateCheck` argument had to be given for the jar executable. Since then the project has bought a certificate and the agent connects without the argument.

As described in chapter 5.5 Proxy, the other problem with Jenkins and EV3 communication emerged from a security concern. It was decided that all the connections to Jenkins, GitLab and OpenLDAP would have to go through a reverse proxy server. Unfortunately, the proxy server could only proxy HTTP and HTTPS packages. No working solution was found how to proxy the traffic from the Jenkins agent with Apache, therefore, a quick and dirty solution was made and the port where the agent was trying to connect was port forwarded with iptables from the proxy server to the Jenkins server.

After figuring out how the Jenkins agent system works and configuring the proxy server the agent setup was easy. The first step to take was to create a new agent to Jenkins. In Manage Jenkins -> Manage Nodes settings window, user has to create a new node. Jenkins asks for some information about the name of the node and such. Special care should be taken when in the node creation Jenkins asks about the node

usage. It might be smart to set this to “leave this node to tied jobs only” setting. This means that the node is only used if its usage is specifically requested in a Jenkins job. The next figure (Figure 8) is a screenshot of node management window on Jenkins.

The screenshot shows the Jenkins interface for a slave node named 'Lego-Gateway'. The node is currently offline. The page provides instructions on how to connect to the slave, including a 'Launch' button and a Java command line. Below this, there is a table titled 'Projects tied to Lego-Gateway'.

S	W	Name ↓	Last Success	Last Failure	Last Duration	
●	☁	ColoRobot-demo	1 mo 28 days - #78	1 mo 0 days - #79	1 min 10 sec	♡
●	☀	ColoRobot-demo-deployment	1 mo 29 days - #6	N/A	15 sec	♡
●	☁	lego-ev3-demo	2 mo 17 days - #19	2 mo 7 days - #20	1 min 54 sec	♡

FIGURE 8. Node management window on Jenkins

In the node management window seen in the figure, users can load the `slave.jar` package by clicking the link embedded in the java command. When the package is loaded the slave can be started by executing the command seen on the node management page. The node is ready to be used by Jenkins when the agent connects. Now when user creates a new job and sets the “Restrict where this project can be run” tag and writes the node name in the text box a job is executed on the newly created node.

As seen in the figure, the ColoRobot-demo deployment job is separate from the other job named similarly. The other ColoRobot job is the building and testing job. That job is triggered by web hook from GitLab always when a new push has been made. After that job has been executed the results from NUnit are checked and if the test failure rate has been inside a failure threshold, only then the deployment job is triggered.

## **GitLab**

There were no major problems at using and configuring GitLab in the continuous deployment chain. GitLab makes it extremely easy to create new repositories and manage user access to it using the web UI it provides. GitLab also has a web hook setting that can be configured for every project separately. The hook can be set to be triggered at push event, tag push event, issues event or at merge request event. The only problem with GitLab emerged when overall read access was denied from anonymous users at Jenkins and that made the web hook to defunct. The problem was solved after Build Token Root Plugin was installed to Jenkins. This allows anonymous users to trigger build jobs using a token.

GitLab allows users to store and enable deploy keys to projects. Deploy keys are normal SSH keys that allow read permissions to repos where they are enabled. A deploy key has to be enabled in a project that is deployed with Jenkins. In Jenkins the private part of the same key has to be saved to credentials and it has to be selected in the project settings when configuring the repository. When Jenkins clones the repository it uses the key which grants a read permission to it and allows an access to the repository.

## **Fabric**

The fabric script that the Jenkins server executes on the agent machine was stored in the same repository as the code that would be deployed. This makes it easy to change something in the script if necessary. The fabric script in this implementation of continuous deployment handled the building, testing and deployment automation. The Jenkins server executes short fab commands on the slave computer and the fabric handles the rest.

As described in the chapter 5.7 Fabric, the scripting with fabric automation framework is very easy. The commands can be run locally or on a remote host. The command execution on a remote host requires that the slave machine has access to it, which is easy to make possible with SSH keys. This way no passwords are required when connecting to the target host. The SSH key is saved in the user's home

directory in the `.ssh` folder. When the key is named `id_rsa` it is always used automatically when trying to access a remote host over SSH. For this to work when Jenkins is controlling the slave, the SSH key has to be saved in the same user's home folder as the one who is running the `slave.jar` package. Other option is to save the SSH key to Jenkins and enable SSH Agent option on the Jenkins job. This makes the Jenkins agent to use a SSH key stored in Jenkins master. The SSH key can be configured from Jenkins web UI.

The building and testing automation were very straight forward to implement with fabric. Using the command line tool `mdtool`, which was bundled with MonoDevelop, it was easy to build the projects by passing a project file for the tool and giving a configuration in which to build the software. The configuration could be debug or release or some custom configuration. The testing was easy to automate as well. After the project containing the tests was built, the tests were executed simply by running `mono testfile.exe -result:'results.xml' -format:'nunit2'`. The result argument states in which file to write the test results and the format parameter tells the NUnitLite to use older format in writing the result file. This was because Jenkins XUnit plugin could not parse the newer version format.

The only problem with Fabric was that the Dropbear SSH server running on EV3 brick did not support SFTP protocol which fabric uses on file transfer. This was quickly fixed by running a command line SCP command instead of using the fabric function. This solved the problem quickly. It could have been solved probably in other ways as well. It might be that changing Dropbear's configuration would also have solved the problem. Changing one line of code on python script was easier.

## **Result**

When all was put together the resulting workflow was as shown in the next figure (Figure 9).

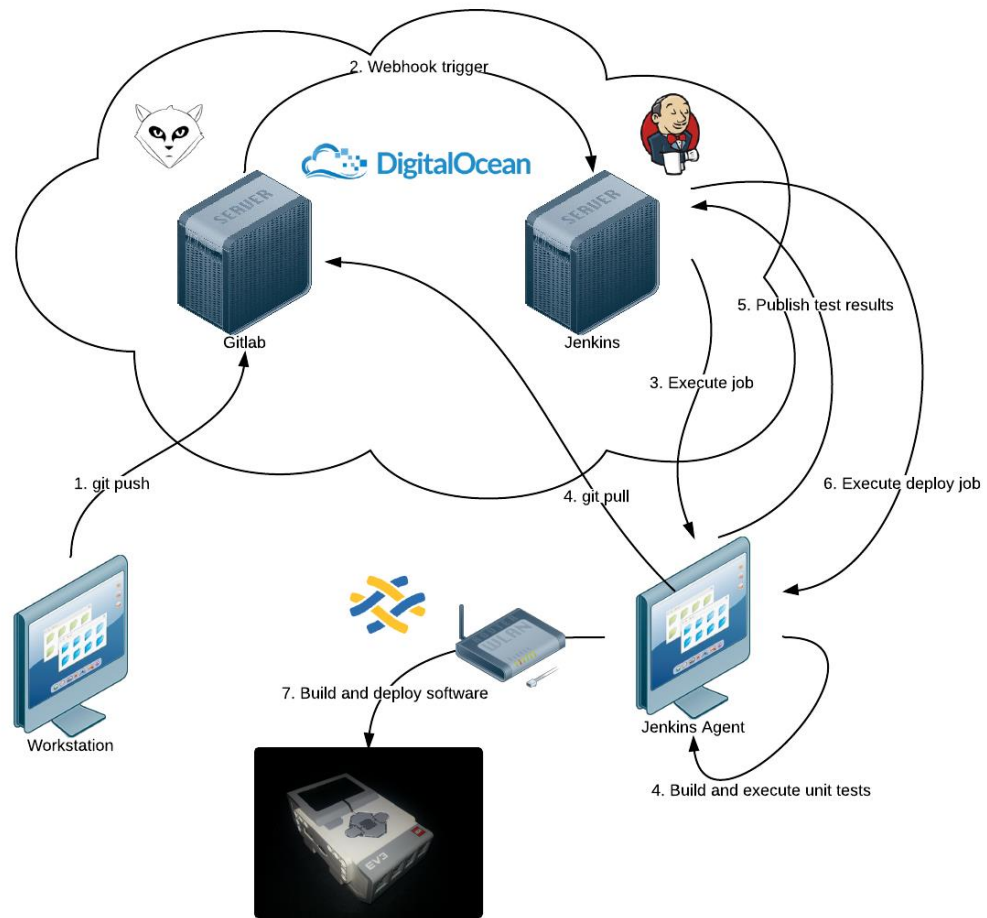


FIGURE 9. Final continuous deployment workflow

In the figure all the steps are numbered and the workflow is from 1 to 7. The first step is when a developer making software to EV3 executes git push command. This command updates the repository existing on a server to match the repository at the developer's computer. This means that all the changes the developer made are copied to GitLab.

The second step is when GitLab notices a new push event to the repository. It triggers the web hook configured to that project. The GitLab makes a HTTP POST request to the URL configured in project settings. If necessary some extra information can be embedded to the URL. The Jenkins Build Token Root Plugin requires the job name and the token to be passed to Jenkins too. This can be made

by appending

“?job=JobnameAtJenkins&token=tokenConfiguredToJob” without the quotation marks.

The third step is when Jenkins notices the POST request to the URL. Jenkins starts to execute the job that is triggered. The job triggered by GitLab is the job that executes the unit tests. In the Jenkins job the job is configured to be executed in the specific slave. Jenkins connects to that slave and clones the repository specified in the job configuration using the deploy key also configured in the job.

The fourth step is that Jenkins executes unit tests on the slave machine. In this case the commands executed are fab commands. Depending on the naming in fab file the first command can be for example `fab build_and_test`. In this case Fabric runs `mdtool` with right parameters to build the NUnitLite project. After that Fabric executes the unit tests by running a `mono` command.

After the unit tests are executed, in the fifth step Jenkins automatically copies the results file to the Jenkins server and starts to parse it. If test failure rate is in the threshold limits, Jenkins triggers the deployment job.

Again in the sixth and seventh step Jenkins connects to the slave machine and executes fab commands. This time the fab command can be for example `fab build_and_deploy`. This command would build the actual project and then deploy it to staging environment.

At this point the actual software is in the staging environment. The figure does not have steps further than that, but the workflow could continue from this. At this point some sort of automatic acceptance tests would be executed to test if the software is ready to be deployed to production. If the test passes there could be a third Jenkins job which is triggered after this job. In that job the software is built and deployed to production. How and where the acceptance tests are executed and how the software is deployed to production can be very project-specific.

## 6.4 Demo

As mentioned before, a demo project was created to test the continuous deployment chain created in this thesis assignment. The demo was presented at Q2 Review in Tampere, at a Demo bazaar event. The demo bazaar was an event where all the companies participating in the Need4Speed program could present their work done since the last review. Many of the companies presented only posters there but JAMK presented many demos including the Lego demo. At the demo bazaar event companies rent a spot from the demo area and setup their demo there. People can then wander around the demo area and watch the demo presentations or read posters.

### **ColoRobot**

The name of the project made for demo purposes was ColoRobot. The name comes from what the robot does. It identifies a color of a ticket inserted in it and prints the color to the EV3's LCD screen. The image of the robot is illustrated in Figure 10.

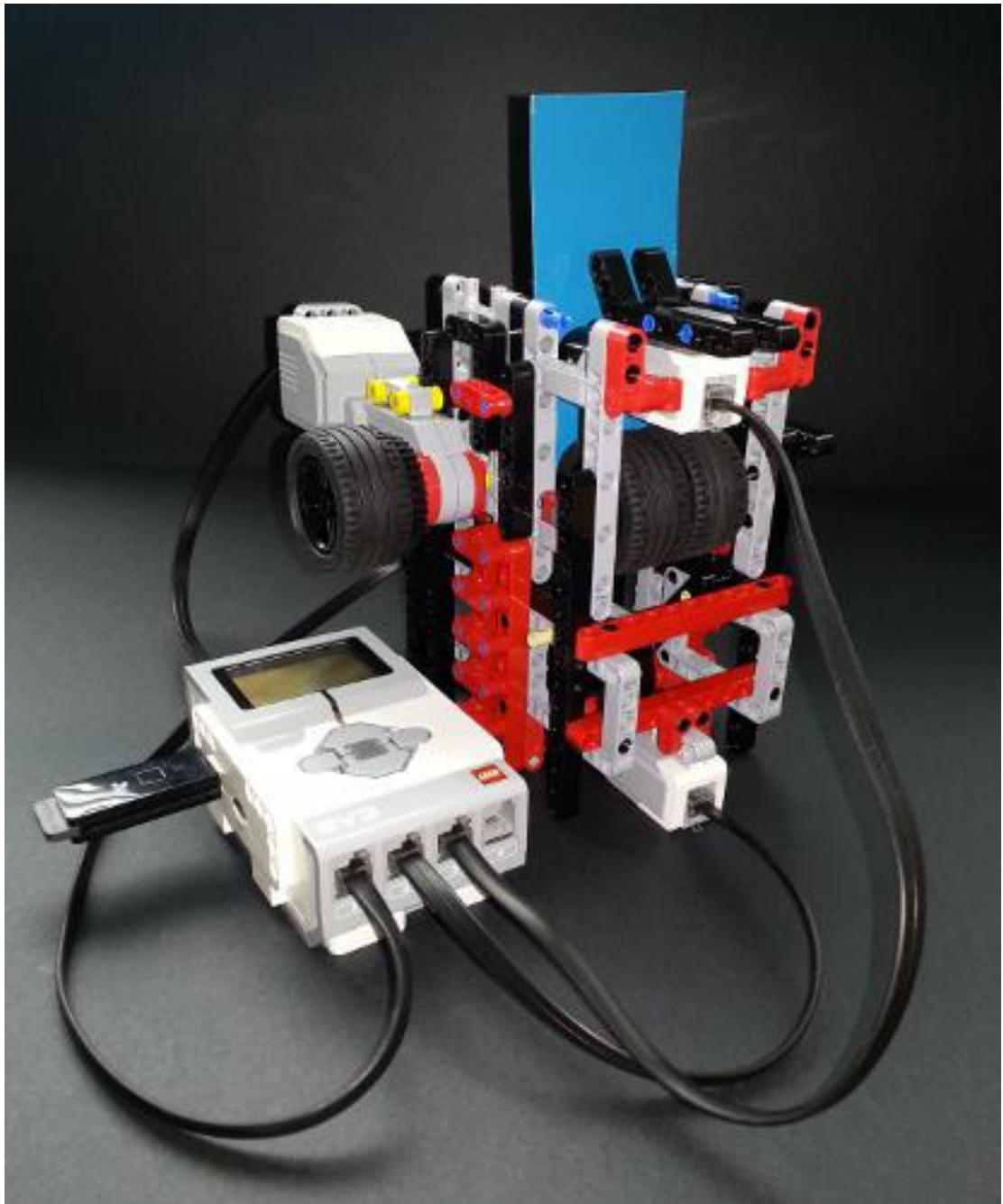


FIGURE 10. ColoRobot

The robot uses three sensors and one motor to operate. The sensors used were IR-sensor, color-sensor and touch sensor. The motor used was one of the two large motors. As seen from the figure, the ticket would be inserted from the top of the robot. On the top part of the robot is the IR-sensor. The IR-sensor detects the distance to the nearest obstacle. This way the ColoRobot knows if a ticket is fed to



the ticket slot. If the distance measured by IR-sensor is less than certain number the motor is started.

The motor is connected to tires with an axel. There is a total of four tires in the structure even if in the image only two are seen. The third tire in the image is only a crank and it is used to turn the motor manually if a ticket is stuck inside the ColoRobot. The ticket inserted in ColoRobot goes between the two sets of tires. The tires are so close together that even when the motor is connected only to two of the tires, all the tires spin. Two of the tires spin clockwise and two of them spin counter-clockwise. This way when the ticket is fed to the robot it pulls the ticket in.

At the bottom of the robot is a touch sensor structure that senses when the ticket is fully in. The bottom of the ticket inserted makes a contact with a building block that presses the touch sensor fully in. When the touch sensor is pressed the robot stops the motor pulling the ticket.

After the ticket is in the robot reads the ticket color with a color sensor. The color sensor is on the other side of the robot and is not seen in the picture. When the color is sensed it is printed to the LCD-screen of the EV3. Then the motor is started again, this time spinning in the opposite direction, which makes the ticket to pop out of the robot.

### **Planning and implementation**

The planning started few weeks before the Q2-review event. There was not much time for planning so the project was implemented very quickly. The structure of ColoRobot is not copied anywhere, the structure was improvised at the same time as the robot was built. The plan was to make something interesting that draws the attention of visitors at the Q2 event.

The software was divided into three different project inside one MonoDevelop solution project. The projects were ColoRobot, ColoRobotDLL and ColoRobotTests. The ColoRobot project is the project that runs the real project and has the logic that makes the robot work as described before. ColoRobotDLL is a library that hides the

low level stuff and makes it easy to write the logic at ColoRobot project.

ColoRobotTests is a project that has the UnitTests and few very simple acceptance tests both implemented using NUnitLite framework. The unit tests also use moq. The test project executes test against the DDL library created in the ColoRobotDLL project.

The workflow was meant to go like this. First Jenkins job executes build commands on the slave machine for ColoRobotDLL and ColoRobotTests projects. Then Jenkins executes command to run the unit tests from the ColoRobotTests project at the slave computer. When the unit tests are executed Jenkins parses the results and triggers the next job if the test failure count is within limits. The next job copies the compiled files from ColoRobotDLL and ColoRobotTests projects to the EV3 brick. On the brick the Jenkins server commands the slave server to run the acceptance tests. These tests are made with NUnitLite too and the result file is copied to slave computer. From there the Jenkins server takes the acceptance testing result file and parses it. If failure count is within limits again, then Jenkins triggers the final job. This job makes the slave computer to compile also the ColoRobot project and then copies it to the EV3 brick and executes it.

Unfortunately, the time given for the implementation did not quite suffice. The unit testing phase was not completed in time because of problems with moq framework. The solution for the moq problem was found on the next day after the Q2-review. There was a missing using statement at the beginning of the file which the MonoDevelop compiler did not point out. There would probably not have been enough time to implement many unit tests anyway so it did not matter so much. The unit testing phase was dropped off entirely and only the acceptance tests were executed when presenting the demo.

At the demo bazaar event, many interested people come and had a look at the demo. Occasionally there were some problems with the internet connection which made the demo run slow when Jenkins did not have connection to the agent

computer all the time. Otherwise the demo went smoothly and there was some positive feedback from people.

## 7 CONCLUSION

All the requirements that were set for this thesis were fulfilled. The focus was on creating a continuous deployment chain for Lego Mindstorms EV3 device and researching ways to make executing unit tests possible on a normal computer. The continuous deployment chain implemented in this thesis uses tools that are in use in many companies and in large-scale open source projects. I believe that the CD chain created here could be adopted to real software projects, because the tools used here are proved to work in real life implementations.

I also believe that the way of making the unit testing possible on a normal computer by mocking interfaces is a good solution. The mocking practice can be adapted for any kind of embedded systems project to some extent. If this thesis only researched way of unit testing Lego Mindstorms EV3 brick by somehow simulating its behavior the result could only be adapted for projects targeting this device. There is probably not many projects that target the EV3.

Of course, testing embedded systems software on a normal computer is not perfect and all-inclusive. Embedded systems have lower specs than normal computers and different processor architectures. Running unit tests on other than the target device does not take into account that. This is not the purpose of unit testing anyway so it does not matter much. The automatic acceptance testing phase is executed on the target machine so this phase tests the software on the right hardware. If this phase is well planned and the tests are comprehensive, this can reveal if the software has problems with the hardware.

This project had the advantage of having an extra layer of abstraction between the software and the hardware. When using mono or .NET, the code is executed with a CLR virtual machine. A CLR is a hypothetical machine on which all the mono or .NET

software is executed. When porting the mono software to a new platform, the CLR component is the one that has to be ported. After the CLR is ported to the platform, all the mono applications run on that platform.

The unit test execution on a normal computer trick used in the thesis assignment takes an advantage on the abstraction that the CLR gives. I do not know what would have changed if the project would have used for example C or C++ as a programming language. These languages do not use any kind of virtual machines. The applications written with these languages are executed very close to the hardware and that way they depend on the hardware they are developed. It might be that the mocking practice for unit testing could be used to some extend even if the programming language was C, C++ or some other low level programming language. However, this was not tested because it was not in the scope of this thesis assignment.

## REFERENCES

Paradigm Change – Delivering Value in Real Time. N.d. N4S project work package introduction. Accessed on 13 June 2014. Retrieved from <http://www.n4s.fi/en/work-packages/paradigm-change-delivering-value-in-real-time/>

Deep Customer Insight. N.d. N4S project work package introduction. Accessed on 13 June 2014. Retrieved from <http://www.n4s.fi/en/work-packages/deep-customer-insight/>

Mercury Business. N.d. N4S project work package introduction. Accessed on 13 June 2014. Retrieved from <http://www.n4s.fi/en/work-packages/mercury-business/>

Chletsos, M. 2012. Continuous Delivery vs Continuous Deployment vs Continuous Integration - Wait huh? Blog post on Assembla Blog page. Accessed on 20 June 2014. Retrieved from <http://blog.assembla.com/assemblablog/tabid/12618/bid/92411/Continuous-Delivery-vs-Continuous-Deployment-vs-Continuous-Integration-Wait-huh.aspx>

Fowler, M. 2013. ContinuousDelivery. Blog post about continuous delivery. Accessed on 20 June 2014. Retrieved from <http://martinfowler.com/bliki/ContinuousDelivery.html>

McNab, S. 2014. Popular Version Control Systems Compared. Blog post about version control software. Accessed on 20 June 2014. Retrieved from <http://www.projecthut.com/version-control-systems-compared/>

Distributed Git - Distributed Workflows. N.d. Page on Git website. Accessed on 20 June 2014. Retrieved from <http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows>

What is a staging environment? N.d. Article about staging environment. Accessed on 21 June 2014. Retrieved from <http://www.programmerinterview.com/index.php/technical-vocabulary/what-is-a-staging-environment/>

Ries, E. 2009. Why continuous deployment?. Article about why to use continuous deployment. Accessed on 26 June 2014. Retrieved from <http://www.startuplessonslearned.com/2009/06/why-continuous-deployment.html>

What is LDAP?. N.d. Article about LDAP. Accessed on 27 June 2014. Retrieved from <http://www.gracion.com/server/whatldap.html>

Git – About. N.d. Page on Git webpage. Accessed on 3 July 2014. Retrieved from <http://git-scm.com/about/small-and-fast>

Eclipse Community Survey 2014 Results. 2014. Article about tools used by Eclipse Community. Accessed on 3 July 2014. Retrieved from <http://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/>

Ohloh - repository compare. 2014. Pie chart about SCM usage in open source projects. Accessed on 3 July 2014. Retrieved from <http://www.ohloh.net/repositories/compare>

Jenkins - Building a software project. N.d. Page on Jenkins website. Accessed on 11 July 2014. Retrieved from <https://wiki.jenkins-ci.org/display/JENKINS/Building+a+software+project>

Jenkins - Distributed builds. N.d. Page on Jenkins website. Accessed on 11 July 2014. Retrieved from <https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds>

Welcome to Fabric!. N.d. Introduction to fabric framework. Accessed on 14 July 2014. Retrieved from <http://www.fabfile.org/>

Microsoft Developer Network - Unit Testing. N.d. Generic explanation about unit testing on Microsoft Developer Network website. Accessed on 15 July 2014. Retrieved from [http://msdn.microsoft.com/en-us/library/aa292197\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx)

xUnit. N.d. Wikipedia article. Accessed on 15 July 2014. Retrieved from <http://en.wikipedia.org/wiki/XUnit>

Unit testing. N.d. Wikipedia article. Accessed on 15 July 2014. Retrieved from [http://en.wikipedia.org/wiki/Unit\\_testing](http://en.wikipedia.org/wiki/Unit_testing)

Mock object. N.d. Wikipedia article. Accessed on 16 July 2014. Retrieved from [http://en.wikipedia.org/wiki/Mock\\_object](http://en.wikipedia.org/wiki/Mock_object)

MonoBrick EV3 Firmware. N.d. Introduction to MonoBrick Firmware on MonoBrick website. Accessed on 17 July 2014. Retrieved from <http://www.monobrick.dk/software/ev3firmware/>

NUnitLite. N.d. Introduction to NUnitLite on nunitlite website. Accessed on 17 July 2014. Retrieved from <http://nunitlite.org/>

moq. N.d. Introduction to moq and source code of moq on GitHub webpage. Accessed on 17 July 2014. Retrieved from <https://github.com/Moq/moq4>

Test-driven development. N.d. Wikipedia article. Accessed on 18 July 2014. Retrieved from [http://en.wikipedia.org/wiki/Test-driven\\_development](http://en.wikipedia.org/wiki/Test-driven_development)

Humble, J. Farley, D. 2010. Continuous Delivery: Anatomy of the Deployment Pipeline. Webpage providing chapter from book: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Accessed on 22 July 2014. Retrieved from <http://www.informit.com/articles/article.aspx?p=1621865&seqNum=5>

Lego Mindstorms EV3. N.d. Wikipedia article. Accessed on 3 August 2014. Retrieved from [http://en.wikipedia.org/wiki/Lego\\_Mindstorms\\_EV3](http://en.wikipedia.org/wiki/Lego_Mindstorms_EV3)

Console Adapter for EV3. N.d. Webpage selling mindstorms accessories. Accessed on 3 August 2014. Retrieved from [http://www.mindsensors.com/index.php?module=pagemaster&PAGE\\_user\\_op=view\\_page&PAGE\\_id=189&MMN\\_position=39:39](http://www.mindsensors.com/index.php?module=pagemaster&PAGE_user_op=view_page&PAGE_id=189&MMN_position=39:39)

Soldaat, X. 2013. EV3: Creating a Console Cable. Article about console access to EV3 device. Accessed on 3 August 2014. Retrieved from <http://botbench.com/blog/2013/08/15/ev3-creating-console-cable/>

Debian on LEGO MINDSTORMS EV3!. N.d. Homepage of the ev3dev project. Accessed on 7 August 2014. Retrieved from <http://www.ev3dev.org/>

Unity (game engine). N.d. Wikipedia article. Accessed on 7 August 2014. Retrieved from [http://en.wikipedia.org/wiki/Unity\\_\(game\\_engine\)](http://en.wikipedia.org/wiki/Unity_(game_engine))

Neely, S. 2013. Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). Agile Conference (AGILE), 2013. IEEE.

Robot Framework – Introduction. N.d. Homepage of the Robot Framework project. Accessed on 9 August 2014. Retrieved from <http://robotframework.org/#introduction>

OpenLDAP. N.d. Homepage of the OpenLDAP project. Accessed on 9 August 2014. Retrieved from <http://www.openldap.org/>