Meri Alho

Automating the Creation and Deployment of New Robot Framework Libraries

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Thesis

20 April 2018

Metropolia

Author	Meri Alho
Title	Automating the Creation and Deployment of New Robot Framework Libraries
Number of Pages	51 pages
Date	20 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Auvo Häkkinen, Principal Lecturer Petri Huovinen, Senior Specialist, Test Automation Architecture

This Bachelor's thesis documents the improvement project of the test automation framework made for Nokia Solutions and Networks. The aim of this thesis was to improve time-consuming processes related to the test automation framework usage by automating some of them. The creation process of new Robot Framework test automation libraries was chosen to be the target improvement, since it included a significant number of manual tasks. Robot Framework is a generic cross-platform test automation framework, and it is widely used throughout the company.

The primary objective of this thesis project was to study options for carrying out the improvement of the Robot Framework library creation process, and to finally develop an implementation for an automated system. The goal of the automated system was to create a new Robot Framework library into the version control system and to generate all needed Jenkins jobs for the new library with minimal user input and effort.

The project automation was based on Jenkins, which is an open source automation server software. The project was divided into smaller sections, which were then divided into subtasks. The two main sections of this project were: creating a template for new libraries with the Cookiecutter tool, and setting up a Jenkins Pipeline to serve as the basis for the automation system. Cookiecutter is a software project templating tool. Testing of the solutions was conducted at all development stages using both unit tests and manual testing.

The system created as the result of this thesis project succeeded well in filling the requirements set for the test automation framework improvement. The system is designed to be easily modifiable, reusable and extendable for future use.

Keywords	Robot Framework, continuous integration, Jenkins, Python packaging, Cookiecutter, Gerrit

Metropolia

Contents

Glossary

ATDD	Acceptance Test-Driven Development. This approach to software development involves the viewpoints of the customer, the development and the testing teams collaborating to create sufficient acceptance tests for the software to verify that the system functions as intended. May be also referred to as Story Test Driven Development (STDD). (Agile Alliance 2017a.)
CLI	Command line interface allows a user to access a computer program through a text-based interface, such as a shell.
CI	Continuous Integration. A practice where code changes are integrated into the version control repository's main branch and tested as early and as often as possible by automated machinery. An essential part of continuous integration is usually automated testing, where tests are run in a repeatable way without the need of manual triggering. (Radigan 2018.)
CRL	Common Robot Libraries. The name used for Nokia internal Robot Framework test libraries.
DSL	Domain-specific language. A programming language that is designed to be used only in a specific application domain. (Wikipedia 2017b.)
Git	A distributed version control system.
Jinja2	A templating language for Python.
Jython	Jython is an implementation of the Python language for the Java platform (Jython Wiki 2018).
PBR	Python Build Reasonableness. A Python library that helps manage Python *setuptools*-based packaging (OpenStack 2018).
PyPI	Python Package Index. A repository of software for the Python programming language (Python Software Foundation 2018).

SSH	The SSH protocol, derived from words Secure Shell protocol, is a method with which one computer can securely login on another remote computer. The protocol can be used for example for authentication and secure file transfer. (SSH Communications Security Inc. 2017.)
TA	Test Automation. The testing of software done with the help of an automated testing tool so that no human interaction or input during testing is required.
TDD	Test-Driven Development. A style of programming, in which the development is divided into three separate phases: coding, testing and refactoring. These phases are executed consecutively, beginning with writing the unit test, which stipulate the requirements the code to be written must fulfill. After that, the code, which is just enough to fill the given requirements, will be written. The final phase is refactoring the code. These steps are done repeatedly and the result of this practice should be full unit test coverage. (Agile Alliance 2017b.)
tox	A generic virtual environment management and test command line tool for Python made for easing packaging, testing and release process of Python software. (Holger et al. 2018a.)
UI	User interface. The system through which the user controls a computer program, an operating system or other appliances. In software usually either graphical or text-based.
VM	Virtual machine. A virtual instance of an emulated computer system, that behaves like a separate computer system.
YAML	YAML ain't Markup Language. A human-readable data serialization standard with minimal block formatting and multiple ways to define the data. (Evans 2016.)
Zip	An archive format of data that is used for data compression and encryption.

1 Introduction

The employer for this thesis was Nokia Networks, which is a subsidiary of Nokia Corporation. The purpose of this bachelor's thesis was to research, plan and implement a solution for a user-friendly way of creating a new Robot Framework test automation library.

The Nokia's Robot Framework test automation libraries are available company-wide so the style and structure of the libraries should be uniform. The existing solution for the creation of the libraries was complicated and time-consuming. The solution also involved a considerable amount of manually generating boilerplate code which often resulted in non-uniform structures and styles between the libraries. The goal was to research which parts of the library creation process could be automated and then plan and implement an automation solution.

Nokia's most important fields of operations include hardware and software used in telecommunications networks, and overall modernization of telecommunications. Nokia's clientele consists mostly of other telecommunications companies. The Nokia Networks subsidiary of Nokia Corporation, formerly known as Nokia Siemens Networks, was created as a result of a mutual venture with Nokia's Network Business and Siemens Communications in 2006. Under the name Nokia Siemens Networks, the company made several acquisitions including the Israeli Ethernet transport systems company Atrica and the wireless network equipment of Motorola. In 2013 Nokia Corporation became the sole owner of the subsidiary. Nokia Siemens Networks was rebranded as Nokia Solutions and Networks and later as Nokia Networks. Now Nokia Networks' main product is mobile broadband technology and it has operations in about 120 different countries. (Wikipedia 2017, Wikipedia 2018.)

This thesis is divided into six sections. The first section in chapter two gives background and introduces the systems, tools and practices used in the process of making this thesis. The second section in chapter three defines the existing problem and presents an objective for solving the problem. Chapter three also includes the planning phase of the proposed solution. Chapter four describes in detail how the implementation of the solution was made and how the solution was eventually achieved. The fifth and final chapter summarizes the process, assesses the implemented solution and presents improvement and further development suggestions for the created system.

2 Project specification

This chapter introduces the Robot Framework which is a generic test automation framework, and describes its working principles (Robot Framework 2018a). Other automation tools and techniques, that were used in the process of making this thesis, are also introduced in this chapter. All the described tools are used in unison and together they form a coherently working test automation framework system.

2.1 Robot Framework

The Robot Framework is a generic cross-platform test automation framework that is designed for acceptance testing and acceptance test driven development (ATDD). Acceptance test driven development focuses on software development based on the requirements and needs of all the collaborating teams which usually include the customer, the development teams and the testing teams. The first version of the Robot Framework was developed for internal use at Nokia Networks in 2005 based on the ideas presented in the Master's thesis of Pekka Klärck. The framework was then later released as open source software in 2008 under the Apache License 2.0. Today the framework is sponsored by the Robot Framework Foundation which consists of companies using and contributing to the framework. (Robot Framework 2018a.)

The concept of the framework is to make software and systems testing easy and more human-readable with a tabular, keyword-driven syntax. A simple example of the usage of this syntax can be seen in listing 1. Robot Framework test files can be written in plain text, HTML, reStructured Text or tab-separated value format and use different suffixes including .txt, .html, .tsv, .rst or .robot. The framework can be used for multiple types of testing, including device, software systems and protocols testing. The testing can be done via various interfaces, for example graphical user interfaces (GUI) and application programming interfaces (API). A graphical user interface is an interface through which the user can interact with a computer system or a program using graphically rendered visual elements. An application programming interface is a set of defined methods through which different software components can communicate. (Robot Framework 2018a.)

```
*** Settings ***
Documentation     Example test cases using the keyword-driven testing
...               approach.
Library           CalculatorLibrary.py

*** Test Cases ***
Push button
    Push button    1
    Result should be    1

Push multiple buttons
    Push button    1
    Push button    2
    Result should be    12

Simple calculation
    Push button    1
    Push button    +
    Push button    2
    Push button    =
    Result should be    3

Longer calculation
    Push buttons    5 + 4 - 3 * 2 / 1 =
    Result should be    3

Clear
    Push button    1
    Push button    C
    Result should be    ${EMPTY}    # ${EMPTY} is a built-in variable
```

Listing 1. A simple calculator example of keyword-driven testing (Robot Framework 2016).

Usually Robot Framework libraries are implemented with Python programming language, or when running Jython they can be implemented with Java programming language. Jython is an implementation of the Python language for the Java platform. The libraries can also be implement with C programming language when using a specific application programming interface. Libraries created with these native Robot Framework languages can also act as wrappers for functionality created with other programming languages. The simplest way of implementing a new framework library is to create a module or a class in Python or Java with methods that map directly to keyword names. (Jython Wiki 2018, Robot Framework 2018b.)

Listing 2 has a simple example of how new Robot Framework library keywords can be implemented with the Python programming language. In the example in listing 2 the module MyLibrary.py contains two functions. These function names are resolved into keyword names in a Robot Framework test case to find the right method or function that is implementing the keyword. Keyword names are case-insensitive and do not take spaces or underscores into account. Listing 3 has an example of how these new keywords introduced in listing 2 can then be used. The used libraries must be defined in the

settings portion of the robot file so the Robot Framework knows where to look for the implementation of the keywords. (Robot Framework 2018b.)

```
def hello(name):
    print("Hello, %s!" % name)

def do_nothing():
    pass
```

Listing 2. Example Python library implemented as a module in the MyLibrary.py file (Robot Framework 2018b.)

```
*** Settings ***
Library    MyLibrary

*** Test Cases ***
My Test
    Do Nothing
    Hello      world
```

Listing 3. An example how the keywords introduced in listing 2 can be used. (Robot Framework 2018b.)

Multiple supporting tools to ease Robot Framework test editing, running, building and other tasks are available. Many of these tools are published as open source software and some of them are integrated in the Robot Framework itself. Both the core framework and Nokia's internal testing libraries are implemented using Python. (Robot Framework 2018a)

Robot Framework testing is extensively used in Nokia and a compilation of testing libraries, named Common Robot Libraries (CRL), is maintained internally by a test automation team. These libraries are for internal use only and they are not published outside the company at this point. At Nokia the Robot Framework is commonly used in acceptance testing of telecommunication server products and solutions.

All Nokia employees are free to use and create new keyword libraries and add them to the Common Robot Libraries collection for common use. The threshold for doing so, however, is fairly high because of the multi-phased process. As a result, the form and structure of the created libraries are often not uniform which complicates the usage of multiple different test libraries together.

2.2 Code review and continuous integration

In a big corporation such as Nokia quality control of software development is essential. Controlling and maintaining the quality of products can be a difficult task when the volume of contributors in software projects is substantial. For this purpose there are code review and continuous integration tools in use to assure the quality of the produced code and to control its input flow into repositories.

The tools used in the Common Robot Library case are Jenkins continuous integration and continuous delivery tool, and Gerrit distributed version control (Git) repository and code review tool. Continuous integration automates the testing and other verifications either when changes are made to the code or at other, preferably frequent intervals. This is done so that any errors or bugs in the code can be detected at an earliest possible phase. Continuous delivery is a software development method which extends on continuous integration. It focuses on a new software build being available for release at any given time. Git is a version control system that tracks changes in project files and stores the project in a remote repository. Both Jenkins and Gerrit are easily extensible tools through various plugins and as such they are highly modifiable to meet the needs of a software development team. The two tools function cooperatively, namely when a code modification is submitted to Gerrit, it triggers a designated Jenkins continuous integration job which runs unit tests and other specified checks.

2.2.1 Jenkins

Jenkins is an open source software automation server that is written in the Java programming language. Originally Jenkins project was developed under the name Hudson, but after a dispute with Oracle Corporation and the principal project contributors in 2010, it was separated from the Hudson project and its name was changed to Jenkins. Now Jenkins software is released under the Massachusetts Institute of Technology (MIT) License. The MIT License is a highly permissive software license that lets the user make changes to the software and use it as they see fit. All software under the MIT License is free to use and the license gives the software users broad copying and distribution rights. (Techopedia 2018.)

Jenkins can be used in multiple different ways and is highly modifiable with its plugins, most of which are made by users themselves and distributed under open source licenses. Jenkins can be accessed and is easily manageable via its user interface or a command line interface (CLI). A command line interface is a text-based interface through which a user can access computer software.

Jenkins offers multiple kinds of jobs and projects that can be defined for continuous integration and continuous delivery purposes. Different projects have different configurations and purposes and many of these projects require certain plugins to work. The Freestyle project is the most commonly used test automation job in continuous integration. The view for creating a new item in Jenkins can be seen from figure 1. (Jenkins 2018a)

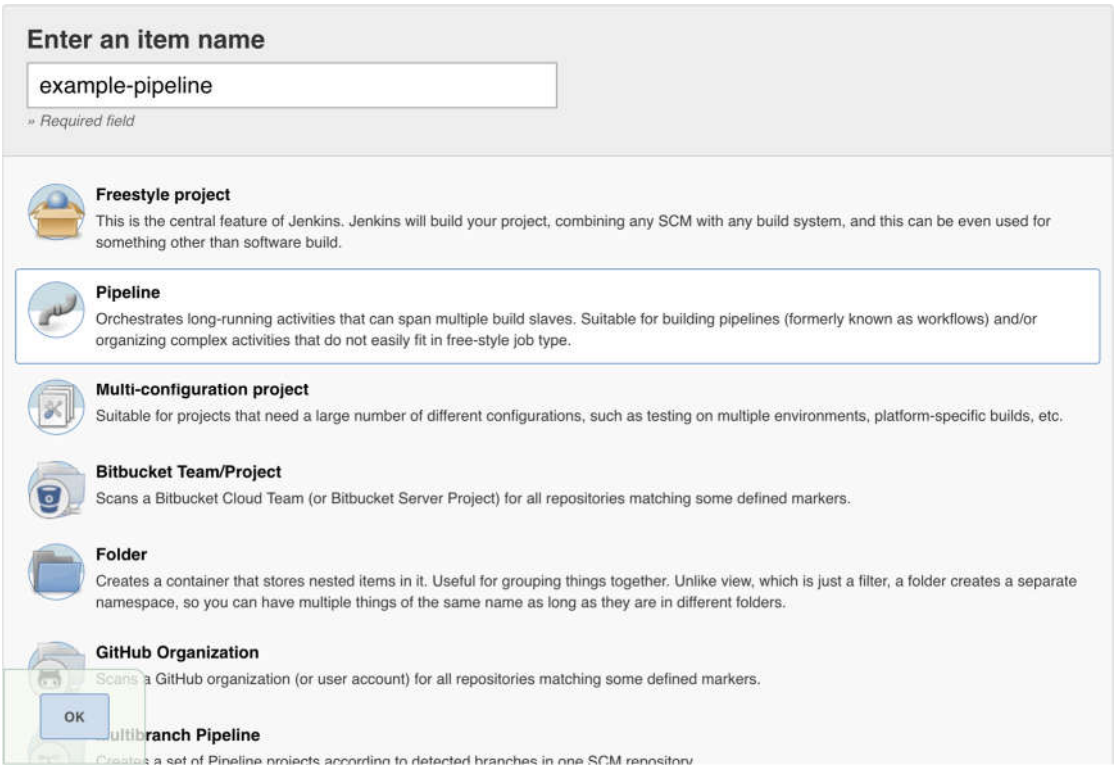

Figure 1. Different project items in the classic Jenkins user interface. (Jenkins 2018a.)

In Common Robot Libraries Jenkins is used as a simple continuous integration automation server, usually housing Freestyle projects and some Pipeline and other projects. Most commonly a Jenkins job listens to a specified Gerrit repository that houses a Common Robot Library so when a code change to the specified library repository is submitted to the Gerrit version control the Jenkins job is triggered automatically. This Jenkins job first retrieves the code from the Gerrit version control and then runs the unit tests that are included in the library. The Jenkins job performs some predefined static checks to

the code and generates the documentation. Usually also a code coverage report is generated, which means that the degree at which the unit tests execute lines of the source code is measured.

Static code checks analyze the source code and check for coding style violations, syntax errors, bugs and bad coding practices without the need for executing the code in any environment. Static code checks enforce the uniformity of coding practices and keep the source code base cleaner since it compels the users to adhere to a certain set of predefined rules. Pylint static analysis is in use for analyzing the Common Robot Libraries source code. Pylint is a Python source code analyzer. Other linting programs are also available for Python and of course other programming languages have their own linter programs. (Logilab et al. 2017)

2.2.2 Gerrit

Gerrit is a free to use, web-based code review and Git repository management software distributed under the Apache License 2.0. The Apache License 2.0 is a permissive open software distribution license written by the Apache Software Foundation. This highly permissive license allows the use of the software for any purpose, modification of the software and distribution of the modified software. All software under the Apache License 2.0 are free to use. (Apache Software Foundation 2017.)

Usually in a collaborative software development environment projects are housed in either Git or some other version control repository. This is called the authoritative repository, which means it is the copy of all files that are included in the project. This authoritative repository is where all developers fetch the project from and push their changes to and it is generally the place the continuous integration machinery, such as Jenkins, uses as a source as seen in figure 2. Gerrit provides an additional layer to this regular Git repository function known as the pending changes layer. Developers still use the authoritative repository to fetch the project, but instead of pushing changes into it the changes are pushed into the pending changes storage as seen from figure 3. The pending changes must then be reviewed and approved before they can be further pushed into the authoritative repository. (Gerrit Code Review 2018a.)

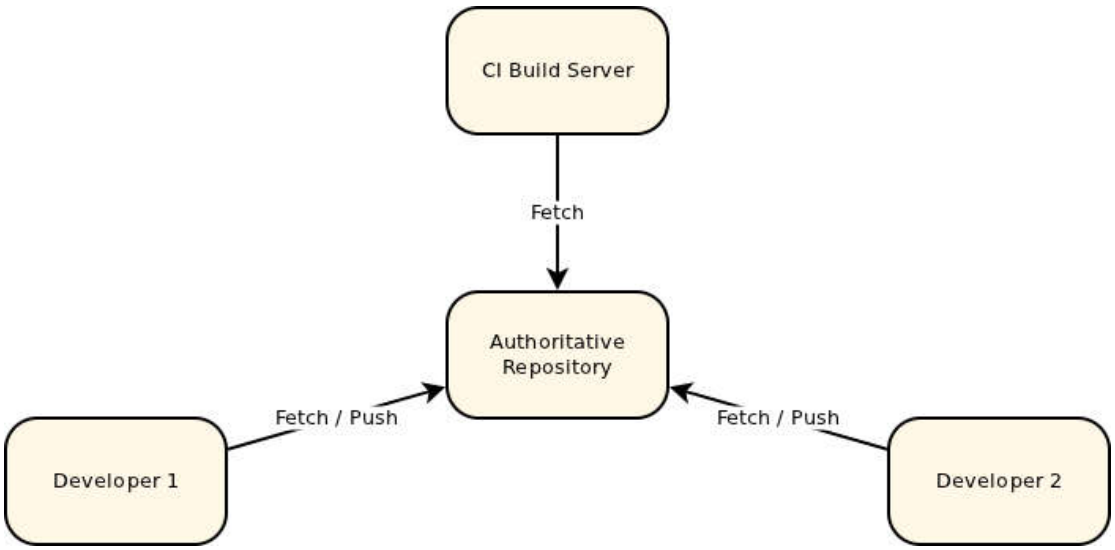

Figure 2. Regular central repository model. (Gerrit Code Review 2018a.)

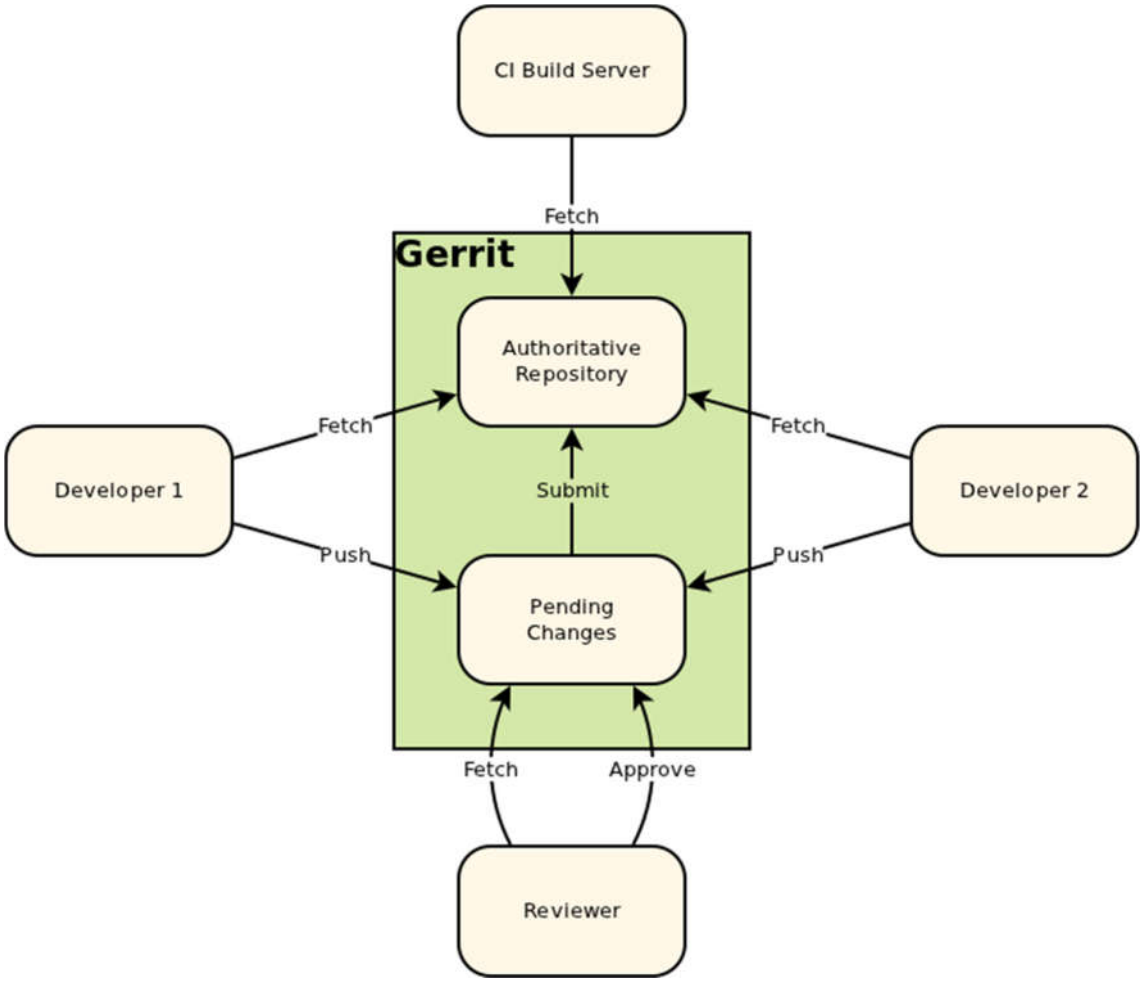

Figure 3. Gerrit repository structure and workflow. (Gerrit Code Review 2018a)

Gerrit makes contributing to a Git-based project repository easier by allowing any authorized user to submit changes into the repositories it houses. This removes the need for a project maintainer to merge all approved changes manually into the master branch. The permissions for each user group are of course configurable since Gerrit has a powerful, user group based access control model. After a change has been submitted into the Gerrit pending changes storage it can be reworked using the same change review process. This can be done by introducing a special Change-Id into the Git commit message based on which Gerrit can then link different versions of the same change together. Gerrit provides a Git commit message hook for this purpose. Git hooks are scripts that are executed before or after certain Git events such as a commit or a push. The Git commit hook will generate a unique Change-Id when the changes are committed. If the hook is not installed and the Change-Id is not manually taken care of the different versions of a same change might end up in different change reviews. (Gerrit Code Review 2018a, Gerrit Code Review 2018b, Hudson 2018.)

Each Common Robot Library has its own Jenkins Freestyle job dedicated for pending code changes submitted to the Gerrit version control. These pending code changes, seen in figure 3 as pending changes, are in the pre-merge stage which means that they are not merged into the authoritative repository of the version control yet. Rather they are still in their own respective branches in the Gerrit pending change storage. Figure 3 shows the repository structure and communication pattern of Gerrit and the continuous integration build server, which in this case is Jenkins.

The Jenkins pre-merge job that is triggered when a code change is submitted to Gerrit sends a verification-vote to Gerrit. The vote value depends on the outcome of the job run. If the Jenkins job completes successfully, like the example pipeline run in figure 4 has done, the verification vote in Gerrit will be set as +1. This means that the code change has gotten a verified status by the continuous integration. In case of failure, aborted run or other non-successful outcome of the continuous integration job the verification vote will be set as –1, meaning that the pending change is not verified. How the verify-label's vote is decided is, of course, configurable in Jenkins via the Gerrit Trigger plugin and in Gerrit by amending user group privileges. Thus, the vote can also be done manually, but in the case of this project the vote always comes from the Jenkins continuous integration machinery. This way no changes without verified unit test and static check success can be submitted to the version control master branch. (Gerrit Code Review 2018a.)

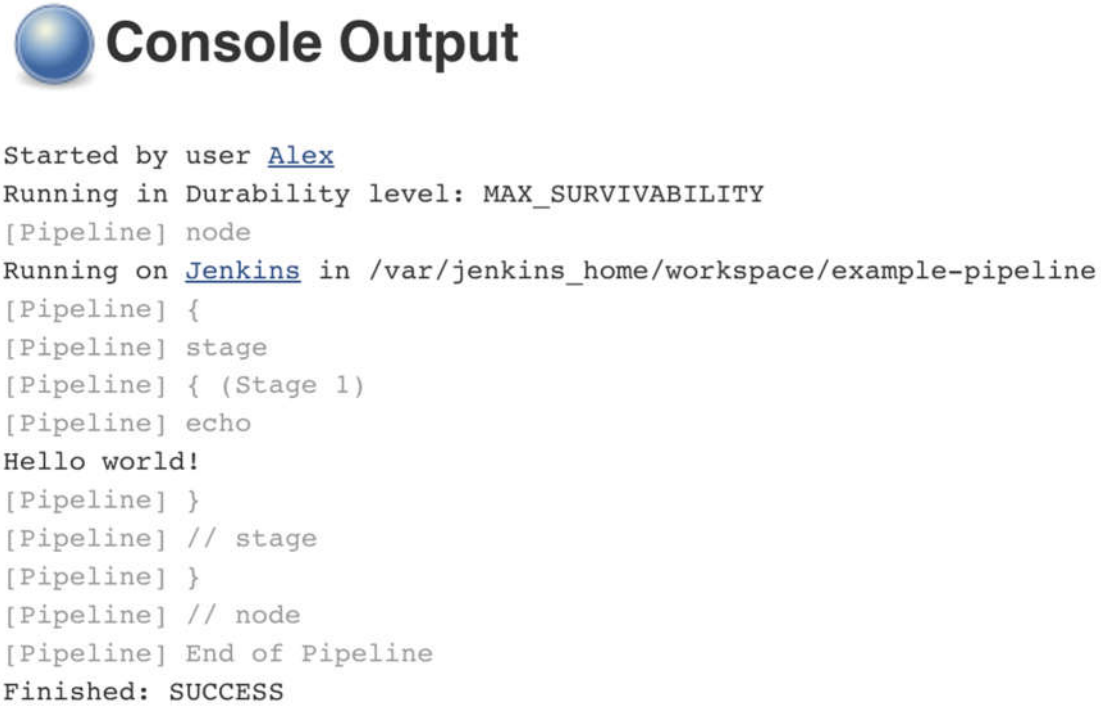

Figure 4. The console output of a simple Jenkins pipeline run example resulting in success. (Jenkins 2018a.)

In addition to the verified-vote the code change needs a code review vote to pass the Gerrit review process. This means that a human must check the proposed changes to see if they conform to the project guidelines and other specifications, and give a vote manually. The code review vote in Gerrit has a larger range of voting options than the verified-vote, which has only two options ranging from –1 to +1. The code review votes range from +2 to –2, with the explanations for different values seen in figure 5.

Figure 5. Code review vote options. (Gerrit Code Review 2018a.)

For the code reviewer's convenience, the Gerrit web-based interface presents the old and new versions of the modified files in a simple side by side view. Here the person who reviews the code changes can make inline comments to specific lines or parts of the code and discuss the changes overall by leaving a general comment. As seen in figure 6, Gerrit shows the new pending change version of the file on the right side showing additions made to the file on a green background colour, and the old version of the file on the left side showing removed contents on a red background. (Gerrit Code Review 2018a.)

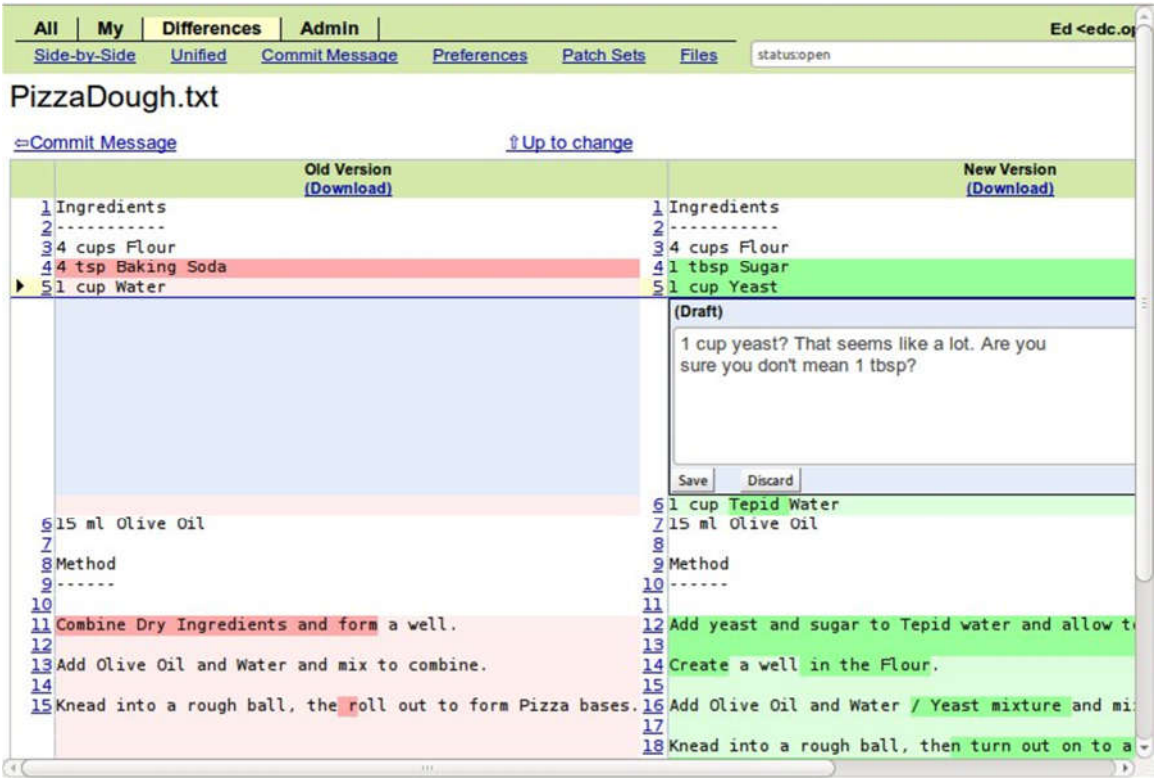

Figure 6. Gerrit side by side view of changes. (Gerrit Code Review 2018a.)

As seen from figure 5, the code review votes +1 and –1 indicate merely the reviewer's level of opinion on the code change. The +2 and –2 votes on the other hand mean either allowing or blocking the change altogether. For a Gerrit change proposal to be accepted it needs at least one +2 vote and it cannot have any –2 votes. The code review votes are not cumulative even though they are numeric values, meaning two +1 votes do not constitute as one +2 vote. If the change is not accepted with a +2 vote it needs to be reworked and go through the verification- and code review processes again. (Gerrit Code Review 2018a.)

2.2.3 Cookiecutter

Cookiecutter is a Python-based command line utility, that creates software projects from project templates called cookiecutters. It creates a software project based on a template and optionally user input. This way all the needed project files are created automatically in one effort. Cookiecutter supports project templates made in any programming language or markup format since it does not execute any of the code inside the template, but rather only fills the defined values inside the template with user input or with values defined in a configuration file. (Roy 2017.)

Cookiecutter uses Jinja2 template engine for its templating needs. Jinja2 is a widely used templating language designed for Python that offers an extensive set of powerful tools for software templating needs. Cookiecutter supports templating directory names and filenames in addition to values inside any files. It also supports creating unlimited nested directories. Listing 2 shows an example of a nested directory with templated directory names and filenames. (Ronacher 2014, Roy 2017.)

```
{{cookiecutter.repo_name}}/{{cookiecutter.repo_name}}/{{cookiecutter.repo_name}}.py
```

Listing 4. Cookiecutter templated nested directory structure.

The parameters that are used in creating the template need to be defined in a cookiecutter.json file as strings, like seen in listing 3. The parameter default values can be left empty by merely leaving the string value empty. All the parameters defined in the file can then be templated anywhere in the project template in the format of *{{cookiecutter.parameter_name}}*. For example, by applying the cookiecutter.json file defined in listing 3 to the directory structure presented in listing 2, the result would be the following directory structure: *example/example/example.py*.

```
{
      "project_name": "Example",
      "repo_name": "example",
      "short_description": "A Cookiecutter example template",
      "version": "0.1.1"
}
```

Listing 5. A cookiecutter.json file contents example.

By default Cookiecutter prompts input for the keys defined in cookiecutter.json. The default responses for these prompts are the default values defined for the keys in the cookiecutter.json file. Alternative values for these keys can also be defined in a configuration

file commonly named .cookiecutterrc. The configuration file uses a YAML-like indented block syntax, as seen in listing 4. YAML, the acronym standing for “YAML ain’t markup language”, is a human-readable data structure standard for all programming languages. The .cookiecutterrc-file is an additional configuration file, which does not replace the cookiecutter.json file in itself, but instead replaces the default values defined in it eliminating the need for user input. (Evans 2016, Roy 2017.)

```
default_context:
      project_name: “Example 2”
      repo_name: “example2”
      short_description: “An example template with .cookiecutterrc file”
      version: “0.2.0”
```

Listing 6. A .cookiecutterrc configuration file contents example.

Templates for Cookiecutter are readily available in different repositories online, but a custom one can also be made if none of the existing ones meet the requirements of a certain project. Cookiecutter supports highly flexibly different storage methods for its templates. Templates can be retrieved either locally, from public and private Git repositories or other online repositories, and even as zip files. (Roy 2017.)

3 Problem definition

This chapter introduces the base problem that this thesis was created to solve. The solution planning and the decided solution for the problem is discussed in detail in this chapter. Also, the different tools used and why they were considered for the solution are explained.

3.1 The problem

Nokia houses a wide collection of Robot Framework libraries on its servers intended for internal test automation use. The collection of these test automation libraries already spans over 20 individual libraries and anyone working for the company is free to generate new ones whenever needed. Generating such a new library, however, is rather time consuming and requires the user to manually write a significant amount of boilerplate code. This fact usually keeps employees from creating such new libraries for common use and often users instead create libraries stored locally and intended only for a single use case. Creating this kind of single use libraries ultimately consumes many times as much working hours as creating a new shared library when the hours put into these kinds of single use libraries are calculated company-wide. One single shared testing library can fit the needs of multiple use cases and thus save the time of multiple employees.

In the rare case that an employee decides to create a new Common Robot Library it is done with a dedicated Jenkins job, that creates a Gerrit repository for the new library and adds only a readme-file to the project. This created readme-file, however, is empty. The user needs to create the entire content for the project themselves almost from scratch, since no configuration files are generated. Even if the user is familiar with the regular Common Robot Library setup and Python packaging needed to distribute them, this step still requires a significant amount of time. Almost all the required files are boilerplate code, which means that the content is basically the same in every created project with only minor differences, such as library owner name and the library name itself. Manually creating this kind of boilerplate code is not beneficial for productivity and is ultimately very inefficient.

Another problem in multiple different users creating and contributing to multiple libraries is that there is bound to be variation in the styles and configurations of the libraries. These differences, especially in the library configurations, can sometimes cause some

of the libraries to be incompatible with each other. This can be a serious issue since dependencies often exist between different Common Robot Libraries. Differences between Python versions, especially versions 2 and 3, can cause the libraries to be incompatible. Both Python 2.7 and Python 3.5 versions are commonly used in the libraries, and the syntax and operations between those two versions is significant. It is enough to obstruct the usage of two different libraries together if the libraries have not been made compliant for both Python versions. Modifying old libraries to be both Python 2 and 3 compliant takes time and effort, which could have been avoided by unifying the libraries from the start. Also, the overall style variation of the libraries can be problematic and trying to create coherent, human readable robot tests with the library keywords can sometimes be difficult.

After a new Common Robot Library is created into the Gerrit repository and its contents is ready, there is still a few steps left in the process. Two new Jenkins jobs need to be created for the new library manually. A pre-merge job, which monitors the changes made via Gerrit and a multi-configuration ci-job with which the new library is tested against any possible dependencies that it might have with any other Common Robot Libraries. Additionally the existing Common Robot Libraries post-merge job in Jenkins must be updated to include the multi-configuration ci-job.

With having to perform all these steps before even getting to make a single unique content to the new library itself it is understandable, that employees prefer the local, single-use test keyword files stored alongside with their tests. Creating a whole Python package project and a lot of boilerplate code for it does not seem a very compelling choice from a single employee's point of view, even though it would be a better option from the perspective of the time consumption of the whole company.

3.2 Objectives as user stories

In this subchapter the objectives for this thesis project are introduced as user stories in table 1. User stories are a common way to define goals for software projects using an agile approach. User stories are short and simple descriptions of a wanted feature, described from the user's or customer's point of view. In planning an implementation for a software some acceptance criteria are normally added, which define when the user story's objective is reached. The user stories for this project are defined in table 1 as user story - acceptance criteria pairs. The first user story in table 1 is the main story, that

is the broadest description of wanted functionalities. The other stories, called here user sub stories, expand the main story to more detailed specifications. (Cohn 2018.)

Project objectives	
User main story	As a Robot Framework test automation user I want a simple and fast way of creating a new Common Robot Library so I can concentrate on the contents and not the framework of the library.
Acceptance criteria	▪ A Jenkins job exists that creates a new Common Robot Library. ▪ A framework for the new library is automatically generated so the user does not have to create parts of the library package themselves. ▪ The new library is automatically added to the Gerrit version control. ▪ The Jenkins job is easy to find and is easy to use. ▪ The user needs to provide only minimal input for the process.
User sub story	As a creator of a new Common Robot Library I want to save time by not having to create boilerplate code and configurations for the library from scratch.
Acceptance criteria	▪ A template for the Common Robot Libraries exists that generates all the boilerplate code when creating a new library. ▪ The template includes all necessary configurations for the library to work. ▪ The template is immediately ready to use and Python packageable.
User sub story	As a creator of a new Common Robot Library I want to save time by having the necessary Jenkins jobs for my new library created automatically.
Acceptance criteria	▪ A Jenkins seed job exists that automatically creates the necessary new Jenkins jobs for the new library. ▪ The new Jenkins jobs are created to the right location. ▪ The new Jenkins jobs are preconfigured. ▪ The user does not have to modify any configurations.
User sub story	As a Robot Framework test automation user I want to be able to create a new Common Robot Library even if I do not know about Python packaging.
Acceptance criteria	▪ The created library structure has all necessary configurations for Python packaging. ▪ The user needs to only know how to run unit tests and other test environments through tox. ▪ The user does not need to make configurations or set an initial version for the library.
User sub story	As a Common Robot Library user I want the new libraries to be uniform for them to be more easily used together.

Acceptance criteria	▪ The new libraries are created with the same configurations. ▪ The new libraries support both Python 2.7 and Python 3.5 for legacy purposes. ▪ The new libraries use tox for testing and virtual environment purposes. ▪ The new libraries have similar structure and naming conventions. ▪ The new libraries are Python packageable. ▪ The new libraries have static code analysis configured and used.

Table 1. Objectives for the project as user stories with their acceptance criteria.

3.3 Planning the solution

This subchapter goes through the reason the tools used in this thesis project were chosen. It describes the way the tools were planned to be used for solving the problem that was described in subchapter 3.1.

3.3.1 Project templating

The first and foremost improvement idea that drove the whole project to be made into a thesis was to somehow template the new Common Robot Library creation. Templating the process would eliminate the need for the user to generate boilerplate code and configure the project from scratch. Templating would also restrict the user input in the Common Robot Library creation process to a bare minimum, which would mean minimal overall variation in outcomes of the library structures. Templated new Common Robot Libraries would have more compatible configurations with any newly created libraries as well as with the existing Common Robot Libraries.

After some research in available software project templating options the Cookiecutter seemed to be best suited for the needs of this project. Cookiecutter is written in Python, it is lightweight and it is very easy to manage. The Cookiecutter package is available in the Python package index (PyPI), which is a repository of software programs written in Python. This eased the use of Cookiecutter in the Common Robot Library development environment, since the package could easily be installed for example by using the tox command line testing tool. (Python Software Foundation 2018.)

Tox is a commonly used tool in Python package test management. With tox the source distribution packaging, installing and testing a Python-based project is made very simple.

Tox environments are configured with a tox.ini file where different virtual environments for running tests can be specified. In the example in listing 7 the tox.ini file has two virtual unit testing environments configured, py27 and py36. When invoking tox from the command line all the environments defined in the *envlist*-portion of the configuration file are run using the configurations and commands defined in the *testenv*-portion of the file. Dependencies defined for each step are installed via a Python package installing tool called *pip*. In the example in listing 7, in addition to the regular test environments, an additional test environment for Cookiecutter is defined. This environment is run only when specifically invoking it with "*tox -e cookiecutter"* command on the command line. It installs the Cookiecutter package and runs the Cookiecutter tool with desired post arguments.

```
[tox]
envlist = py27, py36

[base]
deps =
    pytest
    pytest-cookies

[testenv]
changedir = {envtmpdir}
deps = {[base]deps}
commands = {posargs:py.test {toxinidir}/tests/}
install_command = pip install --no-cache-dir {opts} {packages}

[testenv:cookiecutter]
changedir = {toxinidir}
deps = cookiecutter
commands = cookiecutter {posargs}
```

Listing 7. An example of a tox.ini file, which includes the Cookiecutter command line option.

By having a dedicated environment for the Cookiecutter in the tox.ini file it can be easily used in various environments, such as different Jenkins jobs. It can be installed only for the duration of a Jenkins job run by invoking the environment from the tox.ini file. In Jenkins jobs individual workspaces are created for every job run. After the run has finished, the tools that were installed into the job's workspace are deleted when the workspace is cleaned. This way there is no need to permanently install tools on the Jenkins server itself when the tools are only needed for specific Jenkins job runs.

Cookiecutter also has its own pytest testing tool plugin called pytest-cookies. The pytest-cookies package is also available in the Python package index (PyPI). For tox usage the

plugin should be defined as a base dependency to be installed in every tox run. The dependencies can be seen in listing 7, in the tox.ini file under base dependencies. The plugin eases the testing of a new library creation, which could otherwise be somewhat complicated since a new project needs to be created from the Cookiecutter template under test to ensure the correctness of the template configurations. The pytest-cookies plugin provides a cookies-fixture that is a wrapper for the Cookiecutter application programming interface (API) for generating projects. This way the Cookiecutter tool itself does not need to be invoked directly to test the template. An application programming interface is a set of defined methods through which different software components can communicate. With the pytest-cookies plugin the functionality and the correctness of a template-generated project can be verified using unit tests. The plugin also cleans up the workspace after running the unit tests unless specified otherwise, so no separate management of template-created projects is needed. (Pierzina 2017.)

3.3.2 Continuous integration

With Jenkins being the cornerstone of the continuous integration in the Common Robot Libraries, in addition to a highly modifiable automation server, it was an obvious choice to use it as the basis for the whole project. The old implementation for creating a new Common Robot Library was also executed with a Jenkins job. This existing job was however, as discussed in subchapter 3.1, too inefficient and did not create enough content for the project for easy usage. The job still executed valuable steps in creating a new library straight to the Gerrit version control and thus served as a good starting point for the planning of the new implementation.

The basic idea behind using Jenkins was to create a Jenkins job that could create the new Common Robot Library itself as well as all the required Jenkins jobs for it. A Jenkins Pipeline seemed to be able to handle the all this in a somewhat simplistic way. The Jenkins Pipeline uses a script to execute all operations that are required from it. The Pipeline script can be defined in two different ways as can be seen from the Jenkins Pipeline's dropdown menu in figure 7. The scripts can be written into the Pipeline configuration itself, as seen in the example in figure 8, or they can be housed in a source control management repository such as Git as a file, commonly simply named *Jenkinsfile*.

Figure 7. The options for defining the Jenkins Pipeline.

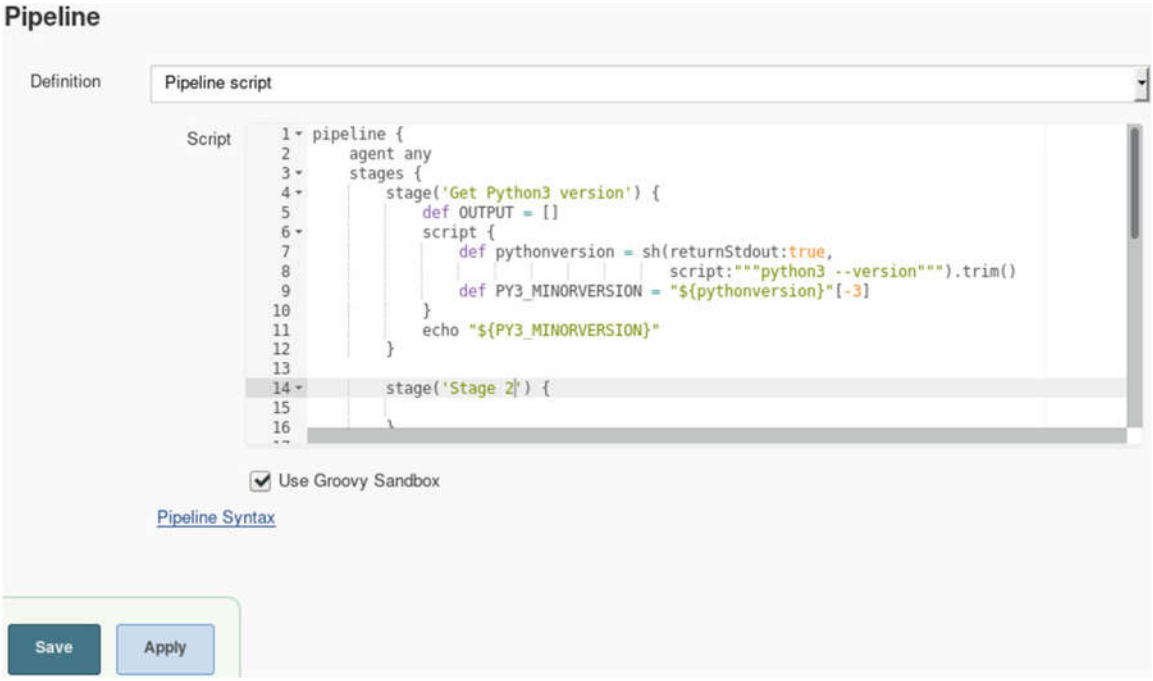

Figure 8. A Jenkins Pipeline script defined in the job configuration.

Using a Jenkinsfile stored in Gerrit version control repository seemed to be a more suitable option for this use case since the pipeline needed to have multiple stages. Creating the script in the small text box of the Jenkins configuration, which can be seen in figure 8, would have been very problematic. Writing the script straight into the Pipeline configuration is better suited for small Pipelines that require only very few stages to be executed. The Jenkinsfile contains all the definitions of a Jenkins Pipeline in one text file and is thus a lot easier to manage using the source control management. The Jenkinsfile itself could be written in the style of a declarative pipeline or a scripted pipeline. The differences between these two techniques are not major, but using the declarative pipeline seemed like a more simplistically managed option of the two.

3.3.3 Source control management

As Gerrit is used as the default source control management tool throughout the Common Robot Library development it was necessary to include it as a part of this project as well. The new library needed to be created into the Gerrit source control repository via the Jenkins job that was planned to handle all the stages of the whole library creation process, as described in the subchapter 3.3.2.

The existing Jenkins job for creating a new Common Robot Library was a good place to start planning the solution for its replacement. The existing Jenkins job implementation included a script with which a new Gerrit repository was created remotely via SSH protocol using the command line interface options offered by Gerrit. The SSH protocol, derived from words Secure Shell protocol, is a method with which one computer can securely log in to another, remote computer. The protocol can be used for example for remote authentication and secure file transfer between computers. (SSH Communications Security Inc. 2017.)

The existing Jenkins script created some empty files, initiated a local Git version control repository and uploaded the files to the new Gerrit repository using Git commands. This script was partially reusable in the project, but it needed to be largely expanded. The new script would have to use the Cookiecutter template and include all the files generated from it into the repository as well as creating the new Gerrit repository itself. Despite the restricted content of the existing script it was a good starting point for getting Jenkins to communicate with Gerrit via an SSH-connection. The new script would have to be a part of the Jenkinsfile, which was discussed in chapter 3.3.2.

3.3.4 Library versioning

The Python build reasonableness, shortened as pbr, is a library for managing Python packaging and versioning in a reasonable and consistent manner. Pbr was first considered for simplifying the version management and packaging needs for the Common Robot Libraries at the start of the project. Pbr, however, turned out not to be the best option for the version management at least when using it as is without any modifications. This was mainly because the version numbering in pbr differed significantly from the convention used in Common Robot Libraries. Although a viable tool for Python packaging it would have had to be modified to meet the needs of the Common Robot Libraries. A test Cookiecutter template was made in which the pbr was used. However, the development

of the template was abandoned as the modifications or wrappers needed for the python build reasonableness tool were evaluated to most likely to be too time-consuming for the timeframe of this thesis project.

4 Implementation

This chapter describes how the planned solution, as explained in chapter three, was implemented for use in a real continuous integration environment. The chapter will specify all the steps and decisions made to reach a working and suitable solution for the project. It also covers how testing the different parts of the project was handled.

4.1 Cookiecutter templating

The first step in this thesis project was to create a custom Cookiecutter template that filled the needs of the Common Robot Libraries. Ready-made Cookiecutter templates for Python packages do exist and are freely downloadable from e.g. GitHub, but the templates differed significantly enough from the basic Common Robot Library package setup that a completely new template was deemed to be the best option. Also, by internally managing the library template the changes made to it could be controlled by the Common Robot Library developer and maintainer team. This new template for the Common Robot Libraries was created from scratch by using the existing Common Robot Libraries as examples. Multiple existing Common Robot Libraries were studied to make a generalized solution that would best suit new libraries and offer the most compatibility with the already existing ones.

4.1.1 Template structure and content

The structure of the template for the new Common Robot Libraries, as seen in listing 9, is a somewhat basic Python package. The template itself needs to be housed inside a Python package, as can be seen from listing 8. Cookiecutter searches for the cookiecutter.json file from the root of the directory and the template directory from the same level as the cookiecutter.json file. The template is referred to from the Cookiecutter command line interface with the package name. In the document tree seen in listing 8, only the *{{cookiecutter._repo_name}}* directory, hooks directory and the tests directory, along with the cookiecutter.json file are related to the template. Other files and directories are used for Python packaging the package itself. Both the top package where the template resides in and the template have similar structures since both follow the packaging guidelines of the Common Robot Libraries. They both use setuptools for package management, tox for virtual environment testing purposes and Sphinx for generating documentation.

```
cookiecutter-crl-template/
├── CHANGES.rst
├── cookiecutter.json
├── {{cookiecutter._repo_name}}/
├── .git/
├── .gitignore
├── hooks
│   ├── __init__.py
│   ├── post_gen_project.py
│   └── pre_gen_project.py
├── MANIFEST.in
├── .pylintrc
├── README.rst
├── setup.cfg
├── setup.py
├── sphinxdocs
│   ├── CHANGES.rst
│   ├── conf.py
│   ├── index.rst
│   └── README.rst
├── src/
├── tests/
│   ├── __init__.py
│   └── test_crltemplate.py
└── tox.ini
```

Listing 8. The cookiecutter-crl-template package. The template directory *{{cookiecutter._repo_name}}* is seen expanded in listing 9.

```
{{cookiecutter._repo_name}}
├── CHANGES.rst
├── MANIFEST.in
├── README.rst
├── setup.cfg
├── setup.py
├── sonar.project-properties
├── sphinxdocs
│   ├── CHANGES.rst
│   ├── conf.py
│   ├── index.rst
│   └── README.rst
├── src
│   └── crl
│       ├── {{cookiecutter._libname}}
│       │   ├── {{cookiecutter._libname}}.py
│       │   ├── __init__.py
│       │   └── _version.py
│       └── __init__.py
├── tests
│   ├── __init__.py
│   └── test_{{cookiecutter._libname}}.py
└── tox.ini
```

Listing 9. The structure of the Common Robot Library Cookiecutter template, which is inside the cookiecutter-crl-template package.

As seen from listing 9, some of the file and directory names in the template package are templated with the Jinja2 style so the names will be looked up from the cookiecutter.json

file or alternatively a user defined configuration file. The template has its own tox configuration file as well as setup files for Python packaging purposes. The Sphinx configuration files have been generated with Sphinx's *sphinx-quickstart* command.

4.1.2 Pre- and post-hooks

Cookiecutter offers a hook script functionality with which hook scripts can be run before and after generating a project from a Cookiecutter template if needed. Support for Python and Shell scripts and for both Unix and Windows systems are built into the Cookiecutter. Python scripts are preferable to shells scripts because of their portability since the Python scripts can be run on any platform. The hook scripts should be housed in the *hooks/* directory in the project root, as seen in listing 10. The files should be named as *pre_gen_project.py* for the script run before generating the project and *post_gen_project.py* for the script run after the project has been generated, with suffixes depending on the script type. (Roy 2018b.)

```
cookiecutter-templatepackage/
├── {{cookiecutter.repo_name}}/
├── hooks
│   ├── pre_gen_project.py
│   └── post_gen_project.py
└── cookiecutter.json
```

Listing 10. Structure with hook scripts.

The Jinja2 syntax can be used in the hooks scripts, just like in the Cookiecutter template itself. With this feature the script that is run before generating the project is perfect for testing the validity of template parameters. In the Common Robot Library case the pre-hook is used for checking the validity of the given library name. The Python pre-hook script, seen in listing 11, checks that the name has the correct prefix and that it does not contain any special characters other than a hyphen. Also, an empty library name or a too short name makes the script exit with a non-zero status code which then stops the project generating process so no files are created. Defining the script's exit status as non-zero is needed if the script should stop generating the project when a condition defined in the script is not fulfilled. If the script exits with an exit code of zero the project is generated from the template so it is important to set correct exit codes.

```
import re
import sys


REPO_REGEX = r'^[a-z]{3}[-a-z0-9]+$'
repo_name = '{{cookiecutter._repo_name}}'


print('Trying to create library with name: %s' % repo_name)


if not 'crl-' in repo_name or 'crl-' is repo_name or not repo_name:
    print("ERROR: INVALID LIBRARY NAME: {name} \nLibrary should be named "
          "in format: 'crl-<libraryname>'".format(name=repo_name))
    print('Stopping creation of library: {name}'.format(name=repo_name))
    sys.exit(1)


if not re.match(REPO_REGEX, repo_name):
    print("ERROR: INVALID LIBRARY NAME: {name} \nLibrary name should not"
          "contain special characters other than '-'. \n"
          "Library name should be in accordance to regular expression: "
          "{regex}".format(name=repo_name, regex=REPO_REGEX))
    print('Stopping creation of library: {name}'.format(name=repo_name))
    sys.exit(1)
```

Listing 11. Cookiecutter's pre-gen-hook.py file

4.1.3 Configuration

The base parameter declaration file in Cookiecutter, as explained in chapter 2.2.3: *Cookiecutter*, is the cookiecutter.json file. This file establishes all the parameters that are to be used in the Cookiecutter template. When invoking Cookiecutter from the command line Cookiecutter will prompt the user for alternative values for the parameters defined in the cookiecutter.json file. This user prompting presented a problem when using the Cookiecutter via the Jenkins scripts. There didn't seem to be a way to ask the user for so much interactive input in a reasonable manner during a Jenkins job run. Even with an existing user-made configuration file such as a .cookiecutterrc file where the user can define new values for either all or some of the parameters introduced in the cookiecutter.json file the Cookiecutter still prompts the user for input. The name .cookiecutterrc for the configuration file is recommended in the Cookiecutter documentation since the extension *rc*, standing for runcom or "run commands", is a common way in Unix systems to name configuration files. Files starting with a dot are hidden files and do not automatically show for the user unless specifically requested and naming configuration files with a preceding dot is a common convention in both Unix and Windows systems. The user is still prompted even with a .cookiecutterrc file present, because the user-made configuration file only replaces the values of the default parameters defined in the cookiecutter.json file, but does not suppress the user prompting in case the user needs to change some of the parameter values. (The Trustees of Indiana University 2018.)

Two relevant options for the parameter input manipulation were presented in the Cookiecutter documentation, as seen in listing 12. Neither of these options, however, were completely suitable for the Common Robot Library use case. The no-input option, as seen in listing 12, does not prompt for parameters, but instead only uses the current content of the cookiecutter.json file. This was not suitable for the use case, because with this option the values set for the cookiecutter.json file's parameters would have to be changed every time a new library was created. It did not seem like a good option to store the changing parameter values in the cookiecutter.json file, which is housed in the version control with the rest of the Common Robot Library template project. The changes would have always had to be added into the version control or alternatively discarded at the end of each Jenkins job run. Generating a completely new cookiecutter.json file every time the Jenkins job was run and not store the file in the version control at all didn't seem like a good option either since the file is essentially the backbone of the Cookiecutter templating. The config-file option was a partially good solution, but as discussed earlier, did not suppress the user prompting and thus did not fix the current problem of user input just by itself.

```
--no-input
Do not prompt for parameters and only use cookiecutter.json file content

--config-file
User configuration file
```

Listing 12. Some of the command line options for Cookiecutter. (Roy 2018a.)

After some research on the internet, a solution for the user prompt problem was found. This solution was not provided in the Cookiecutter documentation at all so it was not easily discovered. By using an underscore as the first character in the parameter names, as seen in listing 13, the parameters would be exempt from the interactive user prompting. Using an underscore as the first character is a common way in Python to declare functions, methods, classes and variables as private and only to be used inside the structure in which they are housed. This rather simple solution combined with a .cookiecutterrc file generated within the Jenkins job run proved to be the best solution. The .cookiecutterrc file would be created inside the Jenkins job workspace during the job run and populated with values derived from the parameters required from the user before starting the job run. The parameters could be manipulated to the wanted form in a shell script inside the Jenkins job. This way the cookiecutter.json file could be stored in the Gerrit

version control and no manipulation of the Cookiecutter base file would be needed. The cookiecutter.json file itself contains default values with which the previously discussed pre-generation hook script fails so no project can be generated with using just the default values in case the user input for the project name would be missing.

```
{
    "_repo_name": "new-cookiecutlibrary",
    "_repo_path": "new.cookiecutlibrary",
    "_libname": "cookiecutlibrary",
    "_author_name": "Library Author",
    "_author_email": "library.author@email.com",
    "_version": "0.1.0",
    "_short_description": "A library created with cookiecutter template",
    "_keywords": "robotframework",
    "_python3_minor_version": ""
}
```

Listing 13. The Cookiecutter.json file with non-promptable parameters.

Jenkins jobs can be configured to be run parameterized in which case the user must input requested values before the job is run. Figure 9 shows the parameter prompt of the new Common Robot Library creation Jenkins job. Some of the parameters have default values set which can guide the user in the right direction or provide values most commonly used. In figure 9 the project name prefix is already provided and the user needs to only add the rest of the library name. The BUILDS_TO_KEEP parameter has a recommended value of 20, so it offers the commonly used amount, but gives the user the option to modify it. These parameters can then be used in the Jenkins job runs as needed with the defined parameter names which can also be seen from figure 9. The use of these parameters is discussed in more detail in the upcoming Jenkins pipeline chapter in section 4.2.

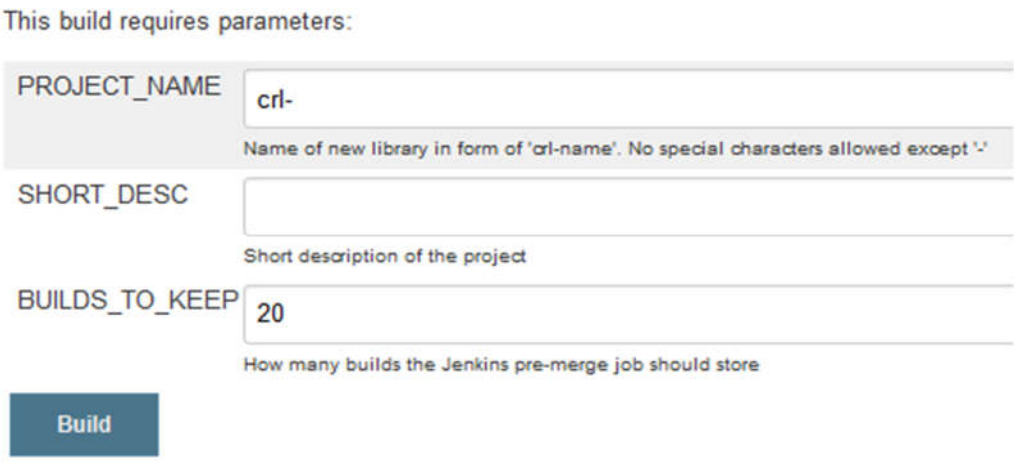

Figure 9. The parameterized build prompt in the new Common Robot Libraries creation Jenkins job.

4.1.4 Testing the Cookiecutter template

Testing the functionality of the Cookiecutter template was done with the pytest-cookies plugin made for the pytest unit testing tool. The unit tests were designed to mainly test four different things about the template:

- all required project files are present and generated with correct names,
- the content of the generated files is correct,
- the pre-hook script works as expected and
- Python packaging of the generated project works.

These four aspects that were designed to be tested are all dependable on each other in some degree since for example Python packaging is not possible if the configuration files do not include correct file paths and other parameters. The use cases of the unit tests could still be divided into these four categories, despite of the internal dependencies of the functionalities. The unit tests were designed to be run in both Python 3.6 and Python 2.7 environments. Python 2.7 had to be included as a testing environment since some of the existing Common Robot Libraries can only be run on Python 2.7 environments. Compatibility with these existing libraries had to be guaranteed in case any dependencies between the existing libraries and a new library were made.

The pytest-cookies plugin provides a *cookies*-fixture that acts as a wrapper for the Cookiecutter's application programming interface for generating new projects. A pytest fixture is a defined, reusable baseline upon which unit tests can be repeatedly executed. When calling the *cookies*-fixture's method *bake()* a new instance of a special Cookiecutter result type is returned. This result instance includes a lot of useful information of the created project, such as an exit code, the exception if one is thrown and finally an object pointing to the rendered project. (Holger et al. 2018b, Pierzina 2017.)

As the cookiecutter.json file was made for the Common Robot Library template in such manner that all projects generated with it would fail in the pre-hook script stage, a custom parameter content for the testing had to be used. Luckily the pytest-cookies plugin accepts a keyword argument that contains parameter - value pairs with which the values from the cookiecutter.json file are replaced. In real use for Common Robot Library creation these values would be created as a .cookiecutterrc file during the Jenkins job run, but for testing purposes using this extra content injection was an easier and more stable

option than dynamically creating new files on every unit test run. The contents of the extra context JSON can be seen from listing 14.

```
extra_context_json = {
    "_repo_name": "crl-cookiecutlibrary",
    "_repo_path": "crl.cookiecutlibrary",
    "_libname": "cookiecutlibrary",
    "_author_name": "Library Author",
    "_author_email": "library.author@email.com",
    "_short_description": "CRL library created with cookiecutter template",
    "_keywords": "test keyword",
    "_python3_minor_version": get_python3_minor_version()
    }
```

Listing 14. Contents of the extra_context JSON for the template unit testing.

```
@pytest.mark.parametrize('exp_dirs', [
    'crl', 'cookiecutlibrary',
    'tests', 'src', 'sphinxdocs'])
def test_dirs_found(cookies, exp_dirs):

    with create_template_in_tmpdir(
            cookies, extra_context=extra_context_json) as result:

        assert result.exit_code == 0
        assert result.project.basename == 'crl-cookiecutlibrary'
        assert result.project.isdir()

        found_dirs, _ = find_all_dirs_and_files(str(result.project))
        assert exp_dirs in found_dirs
```

Listing 15. Extra context JSON and test for checking that all required directories are found.

By parameterizing the tests with the pytest's parametrize functionality multiple expected values can easily be tested without writing individual assertions or unit tests for all the individual desired outcomes. Pytest runs the parametrized test as many times as there are parameters given in the *@pytest.mark.parametrize* block using the parameterized values in the test consecutively. Listing 15 contains a unit test that asserts that all five required directories are present in the project generated from the template using the extra content JSON. A similar test was made for verifying that all required files exist in the document tree and that they are all correctly named.

```
@contextmanager
def create_template_in_tmpdir(cookies, *args, **kwargs):
    result = cookies.bake(*args, **kwargs)
    try:
        yield result
    finally:
        rmtree(str(result.project))


@contextmanager
def inside_dir(dirpath):
    old_path = os.getcwd()
    try:
        os.chdir(dirpath)
        yield
    finally:
        os.chdir(old_path)


def run_inside_dir(command, dirpath):
    with inside_dir(dirpath):
        return subprocess.check_call(shlex.split(command))


def test_make_project_and_run_tests(cookies):
    with create_template_in_tmpdir(
            cookies, extra_context=extra_context_json) as result:
        assert result.project.isdir()
        assert run_inside_dir('python setup.py test', str(result.project))==0
```

Listing 16. The context manager and test for running the tests in the template and testing the Python packaging.

The tests were run so that the templated project was created into a temporary directory so it would be possible to run the tests that were inside the project and test Python packaging of the project. This temporary directory usage could be easily done with a Python contextmanager. The contextmanager yields the pytest-cookies result object and removes the created document tree when the unit test run is complete. These steps can be seen done in the contextmanager-function named create_template_in_tmpdir in listing 16. Even though the pytest-cookies plugin does cleanup after running tests by default it could not be used for the unit tests in this case as is. Running the unit tests that were placed inside the project that was created from the template was not possible without using a specially constructed contextmanager for creating a directory for them to run in. Only by using the pytest-cookies plugin via the contextmanager it was possible to test the python packaging by calling the setup.py file from inside the temporary directory in which the generated project was created, as can be seen from listing 16. The *run_inside_dir* function seen used in listing 16 in the *test_make_project_and_run_tests* unit test uses yet another contextmanager, named inside_dir to run the given command inside the given directory and returns an exit code with which the run completed. The run is successful when the exit code is zero.

All the unit tests created for testing the Common Robot Library Cookiecutter template can be seen from table 2. In the table 2 the tests names, parameterization and the category for which the test is made are listed. Along these tests helper functions, such as contextmanagers, as can be seen in listing 16, and file structure search functions were used to help the testing effort.

Test name	***Parameterized with***	***Test category***
test_files_found	exp_filename	Files present
test_dirs_found	exp_dirs	Files present
test_readme_content		File content
test_changesfile_content		File content
test_toxini_content_py3_minor_version		File content
test_setup_py_content	exp_setup_line	File content
test_with_different_repo_names	extra_context exp_name	Pre-hook
test_pre_hook_fail_with_invalid_repo_names	extra_context	Pre-hook
test_make_project_and_run_tests		Python packaging

Table 2. Tests made for the Cookiecutter template.

The pytest-cookies plugin looks for the Cookiecutter template by default in the current directory so it was necessary to specify a different template location since the unit tests were run in a custom manner inside a temporary directory. The pytest-cookies plugin extends pytest to accept a *--template* parameter for overriding the default Cookiecutter template lookup location. This template parameter had to be added into the tox.ini file to the command with which the pytest unit tests are run.

4.2 Jenkins pipeline

The next phase after finalizing the Cookiecutter template was to create the Jenkins job that would incorporate all the steps needed in creating a new Common Robot Library. This job would not only create the new Common Robot Library project and add it into the Gerrit version control, but it would also create the needed continuous integration jobs for

the project. As discussed in chapter 3.3.2 about Jenkins, the Jenkins Pipeline was chosen for accomplishing all these required steps. The Jenkins Pipeline was configured to fetch a Jenkinsfile script from the Gerrit version control and to use it as a basis for the Pipeline.

4.2.1 Jenkinsfile script

A Jenkinsfile is a text file that defines the actions for a Jenkins Pipeline and is stored in version control. The Jenkinsfile residing in version control provides a multitude of benefits over scripting the Pipeline stages straight into the Jenkins configuration via the graphical user interface. It provides a single source which can be viewed and edited by any member of the project with the correct access rights. The Jenkinsfile also must go through the code review process in Gerrit which keeps the quality of the code on a desired level and gives other developers the possibility to suggest changes. Finally, in the version control all older versions of the file are viewable and restorable in case needed. The Jenkinsfile uses a syntax like the Groovy programming language. Groovy is an object-oriented programming language which is based on the Java programming language platform. (Jenkins 2018b, Tutorialspoint 2018.)

As no suitable candidates for a template for the Pipeline Jenkinsfile were readily available the Jenkinsfile for Common Robot Library creation was built from scratch with the help of the Jenkins Pipeline documentation. The most important feature in a Jenkins Pipeline are the stages to which all functionality can be divided into. Listing 17 shows a simple example of a Jenkinsfile Pipeline script with two stages. This example only prints a string into the Jenkins console, but it shows very simply how the different stages can be orchestrated. One or more *steps* sections are required to be defined inside each stage. Inside a *steps* section one or more steps such as printing into the console in the example in listing 17, must be executed. The *steps*- and *stage* sections can contain multiple other directives and sections which all are more closely explained in the Jenkins Pipeline Syntax manual. (Jenkins 2018c.)

```
pipeline {

    agent any
    stages {

        stage("Hello world") {
            steps {
                echo "Hello world!"
            }
        }
        stage("Hello again") {
            steps {
                echo "Hello!"
            }
        }
    }
}
```

Listing 17. A simple example of a Jenkinsfile declarative Pipeline.

Figure 10. The Create new CRL Library Pipeline stage view in the Jenkins graphical user interface.

In the Jenkins Pipeline script other sections can of course be defined outside the stages portion. Variables and options defined outside the stages portion apply to all stages. Variables and options defined inside a stage apply only to the specific stage. If not defined otherwise when one of the Pipeline stages fails none of the upcoming stages will be executed and the Pipeline run will result in failure. The Pipeline also has a definable *post* section in which the actions executed after running the Pipeline stages can be defined. These post options can be defined differently for different Pipeline run outcomes such as *success*, *failure*, *unstable* and others. Post options most commonly include sending emails about failed or unstable Pipeline runs and cleaning the workspace. Jenkins notifies the user of the failing stage both in the console output and in the graphical user interface. The notification of the failed stage via the graphical user interface can be seen in figure 10.

The Jenkins graphical user interface renders all the Pipeline steps separately visible, as can be seen from figure 10. This view is called the Stage View in Jenkins. Successful Pipeline runs are showed in green and failed runs are showed in red. Furthermore, the stage that caused the script to fail is visibly highlighted, as seen in figure 10. The stage *"Create Jenkins jobs for the library"* in run number 17 can be seen highlighted as the failed stage. Run number 16 is shown with a gray background since on this run Jenkins was unable to run the Pipeline due to syntax errors in the Jenkinsfile. Pipeline runs resulting as *unstable* are shown on a yellow background in the Pipeline Stage View.

The Common Robot Library creation Pipeline was divided into five different stages. In addition to the stages some pre stage actions and post steps were defined. All these five stages execute a single task so in case the Pipeline run fails the user can easily see which stage caused the failure. As can be seen from figure 10, the Jenkins Pipeline Stage View shows seven stages in total. The first stage is in fact the Jenkinsfile being checked out of the version control and the last stage are the previously explained post actions. As mentioned in subchapter 4.1.3, the Jenkins Pipeline was defined to be built with parameters which means that the user must provide values for them before running the Pipeline. These parameters were defined in the Jenkinsfile in the *parameters* section, as can be seen in listing 18. This parameters part of the script is rendered in the Jenkins graphical user interface into what is seen in figure 9 in section 4.1.3. Other general configurations were made for the Pipeline as well. With the *options* directive it is possible to configure the Pipeline from inside the Pipeline script itself. Some configuration options are available via the Jenkins graphical user interface, but these options are very limited and the Jenkinsfile offers a much more versatile configuration manipulation platform. In this case only a few configurations were made via the options section, as can be seen from listing 18. The configured options were: a timeout period for the Pipeline run, disabling any concurrent executions of the Pipeline and adding timestamps to all console output generated during the Pipeline run. Some configurations such as a user access control matrix were made through the graphical user interface. This could also have been configured from the Jenkinsfile, but the user groups and user IDs would have to be hardcoded into the script. Not only that hardcoding such information is generally a bad idea, all the permit configurations would have used a significant amount of lines in the Jenkinsfile. The *environment* section of a Jenkinsfile pipeline, as seen in listing 18, lets environment variables to be set via the Jenkinsfile script as key - value pairs. These variables are usable from inside all the stages and steps when defined outside the *stages* section of the script. They can also be defined inside a stage in which case they are only usable inside that specific stage. The environment variables can be used with an *"env*."

prefix. The parameter variables are used with a *"params*." prefix. The parameter variables can be seen used in the environment section of listing 18. Here the variables gotten from parameterizes user input are manipulated and stored as environment variables. (Jenkins 2018c.)

```
options {
    timeout(time: 15, unit: "MINUTES")
    disableConcurrentBuilds()
    timestamps()
}

parameters {
    string(
        name: "PROJECT_NAME",
        defaultValue: "crl-",
        description: "Name of new library in form of 'crl-name'. No "
                     "special characters allowed except '-'")

    string(
        name: "SHORT_DESC",
        defaultValue: "",
        description: "Short description of the project")

    string(
        name: "BUILDS_TO_KEEP",
        defaultValue: "20",
        description: "How many builds the Jenkins pre-merge job "
                     "should store")
}

environment {
    PROJECT_PATH = "${params.PROJECT_NAME}".replace("-", ".")
    LIBNAME = "${params.PROJECT_NAME}".substring(4).replace("-", "_")
}
```

Listing 18. The pre stage settings for the Common Robot Library creation Jenkinsfile.

The five different stages defined in the Jenkinsfile script were:

1. create a Cookiecutter configuration file,
2. generate a new Common Robot Libraries library from the Cookiecutter template,
3. initialize Git version control for the generated library,
4. create Gerrit repo for the generated library and push the files, and
5. create Jenkins jobs for the library.

The post steps are not taken into account in this stage listing since they differ significantly from the stages defined in the *stages* section of the Jenkinsfile script.

In the first Jenkinsfile-defined stage a Cookiecutter configuration file, named .cookiecutterrc, was locally created. The reasons for creating this file and the usage of the file with the Cookiecutter were explained in more detail in the chapter 4.1.3 about Cookiecutter

configuration. Creating the file was done by invoking a shell script by using the *sh* method of the Jenkins domain specific language, as seen in listing 19. A domain specific language, shortened DSL, is a programming language meant for use only in a particular application domain. As listing 19 shows, the file creation was made with a simple Unix *cat* command that creates the file and adds the content lines into it. As is usually the case, the Common Robot Library Jenkins server uses a Unix-based operating system so all the commands made within the job scripts have to be made in a Unix-style syntax. The parameter- and environment variables that were discussed before and are seen in listing 18 can be seen used in the script in listing 19. The username and user email variables are retrieved from the information of the user who initiated the job by using a Jenkins plugin called Build User Vars Plugin. The Python 3 minor version is gotten by invoking a shell script with the command *python3 --version*. This prints out the current Python 3 version in use in the continuous integration machinery. The output is captured and the minor version from the string is extracted. This information is used in the Cookiecutter template to provide the newest Python 3 iteration in development use for the new Common Robot Library's configuration. This created .cookiecutterrc file is now stored in the Jenkins job's workspace so it can be used in the next stage.

```
sh(script: """
   cat << "EOF" > .cookiecutterrc
   default_context:
      _repo_name: "${params.PROJECT_NAME}"
      _repo_path: "${env.PROJECT_PATH}"
      _libname: "${env.LIBNAME}"
      _author_name: "${USER_NAME}"
      _author_email: "${USER_EMAIL}"
      _short_description: "${params.SHORT_DESC}"
      _python3_minor_version: "${PY3_MINORVERSION}"
   EOF""")
```

Listing 19. The script for creating a .cookiecutterrc file.

The second stage of the Pipeline is where a new Common Robot Libraries library is generated from the Cookiecutter template and stored into the Jenkins job's workspace for further use. This is done by again using the Jenkins domain specific language's shell scripting method. Listing 20 contains the shell script used for this operation. As discussed in chapter 3.3.1 about Cookiecutter a dedicated virtual environment called *cookiecutter* was defined in the tox.ini file of package containing the Cookiecutter template. This way the Cookiecutter could be installed only for a specific Jenkins job run. This virtual environment can now be seen used by invoking it with the *tox* command in the script in listing 20. The location of the configuration file, i.e. the .cookiecutter file created in the previous stage is specified and the version control address from where the Cookiecutter template

itself will be retrieved is given. This script creates the files for the new Common Robot Library project populated with the parameter- and variable values that were introduced previously.

```
sh(script: """
   tox -e cookiecutter -- \\
     --config-file=.cookiecutterrc \\
     git+https://${GERRIT_HTTPS}:${port}/cookicutter-template.git
   """)
```

Listing 20. Creating the project from the Cookiecutter template.

In this second stage the tox virtual environments specified in the newly created project's tox.ini file are stored in a variable as a string. The information about these virtual environments is later needed when creating the Jenkins jobs for the new library. Storing of these variables is done by executing a shell script inside the new project directory and storing capturing the output. The *dir* method of the Jenkins domain specific language executes all wanted actions inside the defined directory. The *tox -l* command lists the virtual environments defined in the tox.ini file into the console where they can be captured from.

```
dir("${params.PROJECT_NAME}") {
      script {
            TOXENVS = sh(returnStdout: true, script: "tox -l").trim()
      }
}
```

Listing 21. Extracting all the environments defined in the tox.ini file to a string.

Stage three simply initializes a Git version control repository for the newly created Common Robot Library folder. The version control initialization is done with basic Git commands that are run inside the new project directory, again using the shell script method provided by the Jenkins domain specific language. After initializing the Git version control for the folder all the files contained in it are added for version control tracking and an initial Git commit is made. The stage does not push the files into a version control repository since there is no repository yet defined for the project. The creating of the version control repository is made in the next stage.

```
script {
  PROJECT_WITH_PARENT = "crl/${params.PROJECT_NAME}"
  GERRIT_SSH = "username@gerrit-url.com"
}

try {
  sh(script: """
     set +x
     ssh -p ${port} ${GERRIT_SSH} gerrit create-project ${PROJECT_WITH_PARENT}
     git remote add origin ssh://${GERRIT_SSH}:${port}/${PROJECT_WITH_PARENT}
     git push -u origin master
     """)

  script {
     currentBuild.description = """<a href='${GERRIT_HTTPS}/\\
       ${PROJECT_WITH_PARENT}'>${params.PROJECT_NAME}</a>"""
  }
}
catch(Exception e){
  echo "Something went wrong! Caught error: ${e}"
  currentBuild.result = "UNSTABLE"
}
```

Listing 22. Creating a new Gerrit version control repository via an SSH connection.

Stage four is dedicated for creating a Gerrit version control repository for the new library and pushing the commit made in stage three into the created Gerrit repository. A shell script connects to the Gerrit host via an SSH connection, as can be seen from listing 22. The SSH protocol, meaning Secure Shell protocol, is a method with which one computer can securely log in to another, remote computer for example for file transferring. Gerrit has a multitude of command line commands through which projects and user groups can be created, downloaded and modified. The *gerrit create-project* command falls under administrator commands which means that when connecting through the SSH protocol the user account used has to have proper administrator rights in Gerrit and a matching SSH-keys stored in the appropriate place in the server and in the Gerrit account. The command creates an empty repository into the Gerrit version control which means that the repository does not have any commits in it yet. Now the repository can be retrieved from Gerrit and an initial commit can be either submitted for review or pushed into straight into a branch. The script used in the Common Robot Library Pipeline, as seen in listing 22, connects the created Gerrit repository as the remote version control repository for the project folder and pushes the project contents straight into the Gerrit repository's master branch. (Gerrit Code Review 2018c, SSH Communications Security Inc. 2017.)

The script is placed into a try-catch block, as can be seen from listing 22. First the try-block's execution is started, but if at any point of the execution an error occurs it is caught and the execution of the block is ceased. The caught exception is defined here as an instance of the general *Exception*-class which is the parent class of all exceptions in

Groovy and Java programming languages. This class catches all possible raised errors. If any errors are caught actions defined in the catch-block are executed. A more specific exception could be defined to be caught, but in this case it is safest to catch all raised exceptions since not all exceptions that could occur are known beforehand. The catch-block of the script marks the overall Pipeline result as *unstable* instead of the default *failure* so the execution of the Pipeline is continued even in case of an error in this stage. This stage is configured this way in case during a previous run of the Pipeline a Gerrit repository was created, but the stage creating the Jenkins jobs for the new library failed or were not executed at all. This way the job can be rebuilt to create the missing Jenkins jobs for a Common Robot Library that is already an existing repository in the Gerrit version control. This also works in the opposite case where the Gerrit repository was not successfully created, but the Jenkins jobs for it were. The latter case could be caused for example by problems with the Gerrit server. The Jenkins job creation in stage five is configured so that if jobs for the given library name already exist they are not modified and no new jobs are created.

Stage five of the Common Robot Library creation Pipeline creates the two necessary Jenkins jobs for the continuous integration of the new Common Robot Library created in the previous stages. This stage uses the jobDSL method which requires the Job DSL Plugin to be installed to Jenkins. As seen from listing 23, this stage seems very simple since the configurations for the new Jenkins jobs are defined in separate files containing Groovy scripts. The names of these scripts are defined in the *targets* parameter of the jobDSL method. The contents and function of these scripts will be more closely examined in the next chapter 4.2.2 *Jenkins Job DSL scripts*. The next parameter seen in the job-DSL method of listing 23, named *ignoreExisting* tells the method that if jobs with the name specified in the target scripts exist they are not to be updated or modified. Only if said jobs do not exist they are then created. The *lookupStrategy* parameter tells how to interpret the names given to the new jobs created from the target scripts. The parameter options are *seed job* and *Jenkins root*. If it is set as the seed job the path where the new job will be created is interpreted as relative to the seed job. The seed job refers to the Jenkins job inside which the jobDSL method is invoked which in this case is the Common Robot Library creation Pipeline. When the lookupStrategy is set as Jenkins root the names given for a generated job will be interpreted as an absolute path. The last used parameter is additional parameters, as seen in listing 23 as *additionalParameters*. This defines a map of global variables which the scripts creating the new jobs can access. (Harmel-Law et al. 2018a.)

```
node("") {
    checkout(scm)
    jobDsl(
        targets: ["pre_jobdsl.groovy", "ci_jobdsl.groovy"].join('\n'),
        ignoreExisting: false,
        lookupStrategy: "JENKINS_ROOT",
        additionalParameters: [
            "pre_jobname": "${params.PROJECT_NAME}-pre-merge",
            "ci_jobname": "ci.${env.PROJECT_PATH}",
            "project_name": "${params.PROJECT_NAME}",
            "gerrit_repo": "${LIB_URL}",
            "library_owner_email": "${USER_EMAIL}",
            "toxenvs": "${TOXENVS}",
            "log_rotation":"${BUILDS_TO_KEEP}"]
    )
}
```

Listing 23. The contents of stage five of the Common Robot Library creation Pipeline.

4.2.2 Jenkins Job DSL scripts

The Job DSL Plugin for Jenkins offers multiple methods for creating new Jenkins jobs, views, folders and configuration files. DSL stands for domain specific language which means that it is a programming language meant to be used inside only a certain application's domain. The scripts using the Job DSL methods are written in Groovy programming language. The methods provided by the Job DSL Plugin are bound to certain Jenkins items and the content for them can be defined in very great detail via the methods' closures. The only mandatory option that has to be defined for the item being created is a name. The Job DSL Plugin has an application programming interface viewer in which all the usable methods are documented. The plugin provides methods for the most common Jenkins use cases, but of course all Jenkins plugins cannot be covered by it, merely because of the sheer volume of available ones. In cases where there is no dedicated method for adding a desired functionality into the Jenkins job a configure block can be used to inject or to modify the information within the config.xml file. Jenkins jobs use the config.xml file for keeping the all job's contents described in one file. Some Job DSL methods have their own configure blocks, but when a dedicated one is not available the job level general configure block can be used. The configure block offers many useful ways to manipulate the config.xml file. The Groovy programming language syntax is used throughout the configure blocks to provide better modifiability. (Harmel-Law et al. 2018b.)

The Job DSL Plugin offers methods for creating different types of Jenkins jobs. The two types used in the scripts for the Common Robot Library continuous integration jobs are Freestyle job and Matrix job. The Freestyle job needed substantially more configuration

overall than the Matrix job, but the Job DSL Plugin had methods for most of the configurations needed. This Freestyle job needed to react to all changes made in the library's Gerrit repository such as incoming reviews, as discussed in subchapter 2.2.1 about Jenkins. Listing 24 shows how the Gerrit trigger is defined with Job DSL methods. The choosing strategy of the source control management section must be set to *gerritTrigger* in addition to defining the trigger itself in a separate *triggers* block. The events section in the Gerrit trigger specifies that the job is triggered every time a draft is published into the repository or a patch set is uploaded to an ongoing review.

```
job("Merge/${pre_jobname}") {
    ...
    scm {
        git {
            remote {
                url("${gerrit_repo}")
            }
            branch("*/master")
            extensions {
                choosingStrategy {
                    gerritTrigger()
                }
            }
        }
    }
    triggers {
        gerrit {
            project("ant:crl/${project_name}", "master")
            configure { node ->
                node << 'serverName'('gerrit')
            }
            events {
                draftPublished()
                patchsetCreated()
            }
            buildFailed(-1, null)
            buildNotBuilt(-1, null)
            buildSuccessful(1, null)
            buildUnstable(-1, null)
        }
    }
}
```

Listing 24. Defining the Gerrit trigger for a Jenkins Freestyle job.

Listing 24 shows the source control management and Gerrit trigger configuration of the Jenkins pre-merge job. The usage of the configure block can also be seen from the listing. Since the Gerrit trigger definition block provided by the Job DSL Plugin does not contain a method or a parameter for separately defining a Gerrit server this has to be done manually via the configure block. The configure block gets a node passed into it. This node represents the context of the configure block from where the config.xml file is parsed. The Gerrit trigger is configured to give Verify votes for the Gerrit review that

triggered the job, as previously explained in chapter 2.2.2 about the Gerrit review process. The Job DSL Plugin offers dedicated methods for defining the Gerrit review votes for different job run outcomes, as can be seen from listing 24. The methods are called buildFailed, buildSuccessful and so on and they can take in two parameters. The first parameter is the numerical value from -1 to +1 given for the Gerrit Verify-label vote. The second one can be a numerical value from -2 to +2 for the Gerrit Code-Review label. The Code-Review values are set to null in this case since as discussed in chapter 2.2.2, a human input is required for the Code-Review process in Common Robot Libraries. A Verify-vote of +1 or -1 is instead given according to the outcome of the job run. (Harmel-Law et al. 2018c.)

```
job("Merge/${pre_jobname}") {
  ...
  configure { project ->
    project / 'publishers' << 'hudson.plugins.labeledgroupedtests.La-
beledGroupedTestArchiver' {
      'configs' {
        'hudson.plugins.labeledgroupedtests.LabeledTestGroupConfiguration' {
          parserClassName 'hudson.tasks.junit.JUnitParser'
          testResultFileMask 'junit.xml'
          delegate.label('unit')
        }
      }
    }
  }
}
```

Listing 25. A publisher defined for the Freestyle job via a configure block.

Listing 25 shows another usage example of the configure block. This configure block adds a publisher that does not have a dedicated Job DSL Plugin method for the Jenkins pre-merge job. The Labeled Test Group Configuration plugin is called by its class path and configurations for it are added. This plugin publishes the Freestyle job run's unit test results from a junit.xml file. This publisher stores the associations between the test result location the parser that needs to be invoked and the label that is applied to the test results. (Yahoo! Inc. 2017.)

The Matrix job is a multi-configuration project with which multiple similar steps are built. It is configured so that build steps do not have to be duplicated as they would be in a regular Jenkins job. The configuration matrix of the Matrix job lets the user to specify which steps need to be duplicated. It creates a multiple-axis matrix of all the builds that need to be created. The multi-configuration job is used in the Common Robot Libraries' continuous integration when a merge in one of the libraries' Gerrit repository is initiated. A continuous integration job runs all the multi-configuration jobs defined for the Common

Robot Libraries to check that any possible dependencies between the modified library and the old libraries are not affected. (Jenkins 2016.)

Since the Job DSL Plugin method provided for the Matrix job has limited internal methods available most of the configurations needed to be done via the configure block. In listing 26 the configuring of the Matrix job's axes with the configure block is shown. The axis method has its own configure block which passes in an axis node to be configured. As explained in chapter 4.2.1 about the Jenkinsfile, the *tox* environments generated for the created library are stored into a variable while running the Jenkinsfile and then passed down for the scripts to use by the jobDSL method's additional parameters list. The *tox* environment string variable can now be seen converted into a list in the beginning of listing 26. The resulting list is then iterated through with a for loop and the items in the list are added into the Matrix job's axis values. The config.xml file generated with this Matrix job creation script is seen in listing 27, displaying the part where the axes values are defined. As can be seen from listing 26, a plugin for a dedicated tox axis configuration exists and is called in the script by its class path. Any version control or other triggers for the Matrix job are not defined in the script since the Matrix jobs are only invoked from a higher level continuous integration job that starts the execution of all the jobs that are configured as having dependencies with the Common Robot Libraries.

```
def toxenvlist = "${toxenvs}".split()

matrixJob("CI/${ci_jobname}") {
    ...
    axes {
        configure { axes ->
            axes << 'jenkins.plugins.shiningpanda.matrix.ToxAxis' {
                name 'TOXENV'
                values {
                    for (env in toxenvlist) {
                        string "${env}"
                    }
                }
            }
        }
    }
...
}
```

Listing 26. Axes defined for the Matrix job by a configure block.

```
<matrix-project>
    <axes>
        <jenkins.plugins.shiningpanda.matrix.ToxAxis>
            <name>TOXENV</name>
            <values>
                <string>py27</string>
                <string>py36</string>
                <string>pylint</string>
                <string>docs</string>
            </values>
        </jenkins.plugins.shiningpanda.matrix.ToxAxis>
    </axes>
</matrix-project>
```

Listing 27. The axes of the resulting config.xml file that is generated with the ci-jobdsl.groovy script.

4.2.3 Testing with Jenkins

The Jenkinsfile's functionality and the proper generating of the new Common Robot library Jenkins jobs had to be tested somehow before the scripts could be submitted for a code review process to be eventually taken into real use. Some unit testing frameworks for testing the Jenkinsfile Pipeline script exist. However, none of these frameworks seemed to be comprehensive enough to include the Job DSL scripts into the testing and some lacked support for methods used in the Jenkinsfile script. The easiest way to test the scripts without a unit testing framework seemed to be by orchestrating an isolated Jenkins instance inside which the scripts could be run. A local instance would allow for running the scripts and modifying them without needing to worry about affecting the actual Common Robot Libraries development environment. A private Jenkins instance was installed onto a personal virtual Linux machine. The scripts were hosted in the Gerrit version control, but stored in a separate branch from the master. This way changes to the branch could be pushed straight into the repository without the Gerrit code review system. The master branch remained empty since changes to it were planned to be submitted through the proper review process.

As all the associated scripts were written from scratch without previous knowledge on the Groovy or Jenkinsfile syntax or their functionality, a lot of trial and failure was involved in the first phase. This is also one of the reasons why the isolated Jenkins server was the best option for initial development since testing of different features could be made without worrying about using too much resources or effecting namespaces on the Common Robot Library continuous integration server. The negative side of using a dedicated Jenkins server for this initial testing phase was the smaller disk space that was available in the personal virtual machines. The smaller capacity prevented installation of some

plugins that were used in the actual continuous integration environment. This was not, however, a major issue since all the most important plugins did fit into the capacity of the virtual machine server and the use of the not installed plugins was not crucial in the testing phase. This initial script development and testing phase was the most time-consuming of all the phases involved in creating the whole system.

The scripts were gradually built up and expanded until all the three scripts were assessed to be in a stage where they could be tested on the real Common Robot Libraries Jenkins server. By testing on the actual Common Robot Libraries server all functionalities offered by the real environment would be available. The scripts were still not configured fully as they were intended to be used in the Common Robot Libraries development since creating an actual Gerrit repository every time the Pipeline was tested was not a good practice. Thus, the Gerrit repository creation stage was omitted from the Jenkinsfile script for testing the Pipeline in the Common Robot Library continuous integration server. Also, the path to which the new Jenkins jobs were created with the Job DSL Plugin was set as *seed job* unlike the value used in the actual Pipeline so test jobs would not populate the pre-merge and ci-job locations. More about the Job DSL Plugin's job paths are explained in chapter 4.2.1 about the Jenkinsfile. This live server testing was done in a folder dedicated only for testing of the Pipeline so other jobs residing on the server would not be affected. As the Job DSL Plugin's path was set to *seed job* the new Jenkins jobs were created into the dedicated testing folder. This second testing and development phase of the Pipeline scripts was significantly shorter than the first phase since the scripts did not need any major modifications at this point. When the testing on the Common Robot Library server was deemed to be sufficient the scripts were submitted to the Gerrit code review process for improvement proposals.

The second testing phase on the real Common Robot Libraries server did run for some time in parallel with the Gerrit code review process since proposed changes for the scripts and other parts of the project also had to be tested to make sure the correct functioning of the finished product. This parallel testing was executed so that a separate version control branch was still used for testing the scripts. The files submitted for the code review and the files in the master branch were kept similar in every other way except for the Gerrit project creation and Job DSL Plugin paths.

4.3 Deployment of the solution

The deployment process for the Common Robot Library Pipeline for creating a new library started with submitting the scripts and the Cookiecutter template to Gerrit code review. The Cookiecutter template and the scripts for the Jenkins Pipeline were housed in the same project in Gerrit. Separate projects for these components could have been made since they are not dependent on each other, but they were connected to the same use case so storing them together was a good solution for future modifications. A pre-merge Jenkins job was created for the Gerrit project so that all included unit tests and other checks such as Pylint would be run every time a change was submitted for a Gerrit review. Unlike regular Common Robot Libraries a post-merge job for this project was not specified since it does not need to be uploaded into the internal Python package index, PyPI, after merging it into the master branch of the Gerrit project. The pre-merge job already checks that the project passes Jenkins gating and send a Verified-label vote to Gerrit. Gerrit also does not let changes with merge conflicts to be submitted into the master branch if used properly through the code review process. A few patch set iterations for the project's code review were made to correct problems and suggested improvements. After all the improvements suggested by a code reviewer were completed the changes were committed into the master branch of the project.

When the code review process was done a Jenkins Pipeline into the correct location in the Common Robot Library Jenkins server was made. The actual Pipeline has the same configurations as the testing Pipeline with the exceptions that the actual Pipeline creates a Gerrit repository for each new library that is generated and the path where new Jenkins jobs are created is set to *Jenkins root* for correct placement in the server's job structure. The old Jenkins job for creating a new Common Robot Library was left as is, but with deprecation notification added into its description.

The final step of the deployment process was to document the functionality of the new library creation system in the officially used channels. The Common Robot Libraries development is documented in the company intranet in the Common Robot Libraries Developers Guide. This page included links into the documentation of the old Common Robot Library creation and some other old information regarding the process. These old pages were updated to match the new system's functions. With these steps executed the new implementation was ready for company wide use.

5 Conclusion

In this final chapter the outcome and effects of the thesis project is evaluated. The value that this project has added to the processes used in the Common Robot Libraries development are discussed. Also, possible improvement suggestions for future development of the project are examined in this chapter.

5.1 Summary of the project

The aim of this Bachelor's thesis project was to implement an improvement solution for the company's test automation framework. The processes used in creating new Robot Framework test libraries was targeted as the subject of this improvement since they included many manual tasks and were overall very inefficient. The goal of the planned improvement was to create an automation system that could create a new Common Robot Library and to generate the Jenkins jobs needed for it, preferably as a one-click system with minimal user input.

The template for the Common Robot Libraries was chosen to be generated with Cookiecutter which is a command line based software project templating tool. The project was automated with the Jenkins automation server. The Jenkins Pipeline item was chosen as the tool to be used in the Jenkins automation since the Pipeline offered versatile options for modifying it to best suite the need of the new automation system. The system was tested thoroughly in all stages of development. Pytest for Python was used for unit testing the Cookiecutter template and manual testing was conducted for the Jenkins scripts.

The project was divided overall into two major parts: the Cookiecutter templating part and the Jenkins Pipeline creation part. These bigger parts were then divided into smaller tasks and the tasks were tracked with a ticket system. The tasks were timed so that first the parts with no dependencies were created and only after those were ready the other parts of the system were developed.

The whole thesis project was done very independently after setting the goals for it with the test automation framework team's lead. The tools and processes used in development of the system could be chosen freely. However, Common practices for the Common Robot Library development were followed in all steps of the project. Of course, help

in problem situations was always provided as needed and the project's progress was reported and assessed in a weekly meeting and reviewing session with the Common Robot Library team. The ticket system was also used for tracking the project's progress.

5.2 Reached objectives

In chapter 3.2 the project's objectives were established as a table of user stories. Here we will reexamine these user stories, as seen in table 3, and see how many of the acceptance criteria were fulfilled by the automation system that was produced as the result of this thesis. All actualized acceptance criteria are marked with a green tick symbol in table 3, and the unsuccessful ones are marked with a red arrow symbol. The overall success of the user story is indicated in the background color of the story itself. As seen from table 3, all but one user stories have succeeded with a hundred percent rate of filling the acceptance criteria. These successful user stories are indicated with a green background and the partially successful user story is indicated with a yellow background.

The only unsuccessful user story acceptance criterion is: *"The user does not have to modify any configurations."* Even though marked as not successful in table 3, this acceptance criterion is still considered fulfilled for the most part since most of the configurations the user needed to make in the old system are now removed and replaced with automatically generated configurations. The reason for highlighting this criterion is that the adding of the new library's continuous integration job into the Jenkins' Gerrit post-merge job still needs to be done manually by the user. Overall, the system that was created as the result of this project succeeded very well in filling the requirements defined in the project's objectives.

Project objectives	
User main story	As a Robot Framework test automation user I want a simple and fast way of creating a new Common Robot Library so I can concentrate on the contents and not the framework of the library.
Acceptance criteria	✓ A Jenkins job exists that creates a new Common Robot Library. ✓ A framework for the new library is automatically generated so the user does not have to create parts of the library package themselves. ✓ The new library is automatically added to the Gerrit version control. ✓ The Jenkins job is easy to find and is easy to use. ✓ The user needs to provide only minimal input for the process.

User sub story	As a creator of a new Common Robot Library I want to save time by not having to create boilerplate code and configurations for the library from scratch.
Acceptance criteria	✓ A template for the Common Robot Libraries exists that generates all the boilerplate code when creating a new library. ✓ The template includes all necessary configurations for the library to work. ✓ The template is immediately ready to use and Python packageable.
User sub story	As a creator of a new Common Robot Library I want to save time by having the necessary Jenkins jobs for my new library created automatically.
Acceptance criteria	✓ A Jenkins seed job exists that automatically creates the necessary new Jenkins jobs for the new library. ✓ The new Jenkins jobs are created to the right location. ✓ The new Jenkins jobs are preconfigured. ➢ The user does not have to modify any configurations.
User sub story	As a Robot Framework test automation user I want to be able to create a new Common Robot Library even if I do not know about Python packaging.
Acceptance criteria	✓ The created library structure has all necessary configurations for Python packaging. ✓ The user needs to only know how to run unit tests and other test environments through tox. ✓ The user does not need to make configurations or set an initial version for the library.
User sub story	As a Common Robot Library user I want the new libraries to be uniform for them to be more easily used together.
Acceptance criteria	✓ The new libraries are created with the same configurations. ✓ The new libraries support both Python 2.7 and Python 3.5 for legacy purposes. ✓ The new libraries use tox for testing and virtual environment purposes. ✓ The new libraries have similar structure and naming conventions. ✓ The new libraries are Python packageable. ✓ The new libraries have static code analysis configured and used.

Table 3. Objectives for the project as user stories with their acceptance criteria.

5.3 Value of the project

The system that was implemented as the project's result significantly decreases the time a Common Robot Library developer needs to use for creating new Robot Framework libraries. It also lowers the threshold for creating new Common Robot Libraries for company-wide use since the user does not need to make any configurations. Since time management is always important in any size companies the time saved from auxiliary processes such as creating new libraries increases the amount of time that is available

for the actual software development. When observing the time savings company-wide it can cumulate into significant amounts. One of the purposes of implementing an easier Common Robot Library creation system was to increase the overall development of new libraries and encourage new users to make their libraries public on the company level. This effect, however, can't be observed yet with the system only just finished.

5.4 Improvement suggestions

As discussed in chapter 5.1 about the unfulfilled user stories' acceptance criterion, adding the new library to the Jenkins post-merge job must still be done manually. At this point there is no way of doing this with methods that the Jenkinsfile scripting offers since the configuration blocks do not allow for appending values into a field that is already populated with values. If no improvement in the Jenkins plugins is implemented for this in the future the option to do this automatically would be to parse the config.xml file generated for the post-merge job by Jenkins and to append the new value with Groovy programming language string manipulation.

As discussed in chapter 3.3.4 about the Python build reasonableness tool it could be added to the Common Robot Libraries for simplifying library version management and for more easily generating the changelog. This would, however, require modifications or wrappers made for the pbr tool. Still it can be considered a viable option for further examination. With using the pbr a new Cookiecutter template configured specifically for use with pbr would also be needed.

References

Agile Alliance. 2017a. Acceptance Test Driven Development (ATDD). Web-page. <https://www.agilealliance.org/glossary/atdd/>. 29.11.2017. Retrieved 25.01.2018.

Agile Alliance. 2017b. TDD. Web-page. <https://www.agilealliance.org/glossary/tdd/>. 29.11.2017. Retrieved 25.01.2018.

Apache Software Foundation. 2017. Apache License, Version 2.0. Web-page. <http://www.apache.org/licenses/LICENSE-2.0 >. 19.09.2017. Retrieved 13.03.2018.

Cohn, Mike. 2018. User Stories. Web-page. <https://www.mountaingoatsoftware.com/agile/user-stories>. Retrieved 13.03.2018.

Evans. Clark C. 2016. YAML 1.2. Web-page. <http://yaml.org/>. 02.12.2016. Retrieved 02.03.2018.

Gerrit Code Review. 2018a. Gerrit Code Review – A Quick Introduction. Web-page. <https://gerrit-documentation.storage.googleapis.com/Documentation/2.14.7/intro-quick.html>. Retrieved 13.03.2018.

Gerrit Code Review. 2018b. Gerrit Code Review. Web-page. <https://gerrit.googlesource.com/gerrit>. Retrieved 13.03.2018.

Gerrit Code Review. 2018c. Gerrit Code Review - Command Line Tools. Web-page. <https://review.openstack.org/Documentation/cmd-index.html >. Retrieved 14.04.2018.

Harmel-Law, Ryan, Sheehan and Spilker. 2018a. Job DSL Plugin – User Power Moves. Web-page. <https://github.com/jenkinsci/job-dsl-plugin/wiki/User-Power-Moves>. 14.04.2018. Retrieved 14.04.2018.

Harmel-Law, Ryan, Sheehan and Spilker. 2018b. Job DSL Plugin – Job DSL Commands. Web-page. <https://github.com/jenkinsci/job-dsl-plugin/wiki/Job-DSL-Commands>. 09.02.2018. Retrieved 14.04.2018.

Harmel-Law, Ryan, Sheehan and Spilker. 2018c. Job DSL Plugin – The configure Block. Web-page. <https://github.com/jenkinsci/job-dsl-plugin/wiki/The-Configure-Block>. 10.04.2018. Retrieved 15.04.2018.

Holger, Krekel and others. 2018a. Welcome to the tox automation project – tox 3.0.0rc2.dev7 documentation. Web-page. <https://tox.readthedocs.io/en/latest/>. 18.02.2018. Retrieved 02.03.2018.

Holger, Krekel and pytest-dev team. 2018b. Pytest fixtures: explicit, modular, scalable. Web-page. <https://docs.pytest.org/en/latest/fixture.html>. 07.04.2018. Retrieved 09.04.2018.

Hudson, Matthew. 2018. Git Hooks. Web-page. <http://githooks.com/>. 07.02.2018. Retrieved 13.03.2018.

Jenkins. 2018a. Getting Started with Pipeline. Web-page. <https://jenkins.io/doc/book/pipeline/getting-started/>. Retrieved 09.02.2018

Jenkins. 2018b. Using a Jenkinsfile. Web-page. <https://jenkins.io/doc/book/pipeline/jenkinsfile/>. Retrieved 10.04.2018.

Jenkins. 2018c. Pipeline Syntax. Web-page. <https://jenkins.io/doc/book/pipeline/syntax/>. Retrieved 10.04.2018.

Jenkins. 2016. Building a matrix project. Web-page. <https://wiki.jenkins.io/display/JENKINS/Building+a+matrix+project >. 16.02.1026. Retrieved 15.04.2018.

Jython Wiki. 2018. General Information. Web-page. <https://wiki.python.org/jython/JythonFaq/GeneralInfo>. 01.01.2018. Retrieved 12.03.2018.

Logilab, PyCQA and contributors. 2017. Web-page. <https://pylint.readthedocs.io/en/latest/intro.html>. Retrieved 09.02.2018.

OpenStack. 2018. pbr - Python Build Reasonableness. Web-page. <https://pypi.python.org/pypi/pbr>. 09.01.2018. Retrieved 17.01.2018.

Pierzina, Raphael. 2017. Pytest-cookies. Web-page. <https://github.com/hackebrot/pytest-cookies>. 13.08.2017. Retrieved 17.01.2018.

Python Software Foundation. 2018. PyPI - the Python Package Index. <https://pypi.python.org/pypi >. 25.01.2018. Retrieved 25.01.2018.

Radigan, Dan. 2018. Continuous integration, explained. Web-page. <https://www.atlassian.com/continuous-delivery/continuous-integration-intro>. 17.01.2018. Retrieved 17.01.2018.

Robot Framework. 2018a. Web-page. <http://robotframework.org/>. 18.01.2018. Retrieved 25.01.2018.

Robot Framework. 2018b. Robot Framework User Guide. Web-page. <http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-test-libraries>. Retrieved 12.03.2018.

Robot Framework. 2016. Web-page / Git repository. <https://bitbucket.org/robotframework/robotdemo>. 29.01.2016. Retrieved 25.01.2018.

Ronacher, Armin. 2014. Jinja. Web-page. <http://jinja.pocoo.org/>. 07.04.2014. Retrieved 13.03.2018.

Roy, Audrey. 2017. Cookiecutter. Web-page. <https://cookiecutter.readthedocs.io/en/latest/readme.html>. 06.12.2017. Retrieved 17.01.2018.

Roy, Audrey. 2018a. Command Line Options. <http://cookiecutter.readthedocs.io/en/latest/advanced/cli_options.html>. 27.03.2018. Retrieved 05.04.2018.

Roy, Audrey. 2018b. Using Pre/Post-Generate Hooks. <http://cookiecutter.readthedocs.io/en/latest/advanced/hooks.html >. 27.03.2018. Retrieved 05.04.2018.

SSH Communications Security Inc. 2017. SSH protocol. Web-page. <https://www.ssh.com/ssh/protocol/>. 29.08.2017. Retrieved 28.03.2018.

Techopedia. 2018. MIT License. Web-page. <https://www.techopedia.com/definition/3287/mit-license>. Retrieved 13.02.2018.

The Trustees of Indiana University. 2018. In Unix, what do some obscurely named commands stand for? Web-page. <https://kb.iu.edu/d/abnd>. 18.01.2018. Retrieved 09.04.2018.

Tutorialspoint. 2018. Groovy Tutorial. Web-page. <https://www.tutorialspoint.com/groovy/index.htm>. Retrieved 10.04.2018.

Wikipedia. 2018. Nokia. Web-page. <https://en.wikipedia.org/wiki/Nokia>. 15.01.2018. Retrieved 17.01.2018.

Wikipedia. 2017. Nokia Networks. Web-page. <https://en.wikipedia.org/wiki/Nokia_Networks>. 23.12.2017. Retrieved 17.01.2018.

Wikipedia. 2017b. Domain-specific language. Web-page. <https://en.wikipedia.org/wiki/Domain-specific_language>. 04.12.2017. Retrieved 17.01.2018.

Yahoo! Inc. 2017. Package hudson.plugins.labeledgroupedtests. Web-page. <http://javadoc.jenkins.io/plugin/labeled-test-groups-publisher/hudson/plugins/labeledgroupedtests/package-summary.html >. Retrieved 15.04.2018.