Master Thesis
Information Science

Radboud University Nijmegen

Evaluating the testing quality of software defined infrastructures

Author:
B. Siebert, BSc
b.siebert@student.ru.nl
Student No. 0839671

Internal supervisor:
Prof. dr. M.C.J.D. van Eekelen
marko@cs.ru.nl

External supervisor:
Prof. dr. ir. J. Visser
j.visser@cs.ru.nl

22nd September 2014

Abstract

Software defined infrastructures are computer infrastructures expressed in computer code. As software defined infrastructures is a young technology the body of scientific knowledge on the subject is small.

This study aims at identification of quality aspects of software defined infrastructure projects by interviewing practitioners in the field. The result is a quality model containing quality characteristics of software defined infrastructures.

During the interviews on quality of software defined infrastructures, testing emerged as one of the most important quality aspects. This study therefore also aims at creating a testing quality model for evaluating the testing quality of software defined infrastructures. This was done by applying a testing quality model made by the Software Improvement Group (SIG) for traditional software to software defined infrastructures and by interviewing practitioners on their opinion of the existing model. The result is an improved testing quality model which is better suited for software defined infrastructures.

Acknowledgements

I would like to take this opportunity to thank the people that have helped me make this thesis and my education possible.

First I would like to thank Marko van Eekelen en Joost Visser for guiding me trough the tough process of creating a master thesis. At moments where I felt that I could not see where I was heading, your advice provided the clarity needed to continue.

I would also like to thank Joost for giving me the opportunity to write this thesis at the Software Improvement Group (SIG). Being a research intern at SIG has been a great learning experience which I think will influence my whole future career.

The members of the SDI team of SIG, namely Thomas Kraus, Wander Grevink and Kay Grosskop together with Miguel Ferreira from Shuberg Philis who allowed me to take a look at their SDI projects. Prior to writing this thesis I knew nothing about SDI. Now I know a little more. Thank you.

The research department of SIG, and especially the other research interns can not go unmentioned. Being around peers on a similar mission has been truly inspiring. I wish you all the best of luck in your careers.

But most of all I would like to thank my mother, my father, my sister and my friends. You enabled me to pursue my ambitions and provided me encouragement when I needed it.

Ben
's-Hertogenbosch
September, 2014

Contents

1 Introduction **1**
1.1 Problem statement 1
1.2 Research questions 3
1.3 Research method 4
1.4 Thesis outline 5

2 Background **6**
2.1 Software defined infrastructures 6
2.2 Infrastructure management using Puppet 9
2.3 Cloud computing 13
2.4 DevOps 15

3 Identification of quality aspects **17**
3.1 Background 17
3.2 Gathering SDI quality aspects 20
3.3 Construction the SDI quality model 25
3.4 Conclusion 29

4 Testing quality of software defined infrastructures **30**
4.1 Background 30
4.2 SIG testing quality model 35
4.3 New testing quality model 45
4.4 Conclusion 52

5 Conclusion **53**
5.1 Results 53
5.2 Future work 56

Bibliography **58**

A Interviews SDI quality characteristics **60**
A.1 Interview #1 60
A.2 Interview #2 62
A.3 Interview #3 63

A.4 Interview #4 64
A.5 Interview #5 65

B Reference topics and transcription **68**

C Application of TQM interview guide **76**

D Test quality model questionnaires **78**

Chapter 1

Introduction

We start this chapter by providing a small background on software defined infrastructures and some of the issues related to it. From this background we formulate research questions which we answer in this thesis. We then explain the research methods used for answering the research questions. We end this chapter by giving an outline for the rest of the thesis.

1.1 Problem statement

In this section we describe some problems surrounding software defined infrastructures. We start by explaining that the application of software defined infrastructures is relatively new. We then explain how the problems that software defined infrastructures solve were dealt with before software defined infrastructures were applied. We end by portraying the current state of the research on the topic of software defined infrastructures.

1.1.1 New area of interest

Software defined infrastructures are computer infrastructures expressed as code. By using configuration management tools and languages it is now possible to automatically provision a computer by application of these infrastructure definitions. In traditional software development one compiles source code into machine code or bytecode to execute it, or inputs the source code to an interpreter which then executes it. When creating software defined infrastructures the source code is a script defining steps a machine has to perform or is a declaration of what the configuration of a computer should be. The scripts can be performed ‘directly’ on the target machine, while the declarations need a intermediating tool that transforms the definitions into actions which the computer should perform.

Open Source configuration management tools such as Ansible[1], Chef[2], Puppet[3] and Salt[4] provide a means to provision many computers by writing code defining the desired configuration for these computers. These definitions describe, for example, that a particular application should be installed and running, or that the operating system should be configured in a certain way. The configuration management tools are made to be independent of the operating system they are run on by abstracting operating system-specific details away, such as which package management system the operating system uses. In practice however, when developing the infrastructure using configuration management tools, the developer still has to develop code with certain operating system in mind because of the varying dependencies of the operation system, such as where the operating system stores its applications or configuration.

Before the adoption of configuration management tools, system administrators would provision computers by creating images of fully installed machines and by writing scripts to deploy or update applications. The problem with these methods is that they are operating system dependant, inflexible, error prone and therefore limit the amount of computers that a single system administrator can manage.

The emergence of open source configuration management tools, combined with virtualisation and cloud computing has led to a wide adoption of software defined infrastructures by the IT-industry. Companies such as PayPal[5] and Spotify[6] leverage configuration management tools to automate their IT infrastructures. Although many companies use software defined infrastructures to automate their infrastructures, the academic community has not yet picked up software defined infrastructures as an area of research.

1.1.2 Infrastructure creation as a software development process

As described, one of the features of software defined infrastructures are that they are expressed in computer code. This means that the creation of computer infrastructures can now be treated as a software development process. The development of software is traditionally the field of software engineering, where techniques are devised to create software in an efficient and robust manner.

Software engineering can be considered an immature engineering field in comparison to other engineering fields. Yet while software engineering can

[1] http://www.ansible.com/
[2] http://www.getchef.com/
[3] http://puppetlabs.com/
[4] http://www.saltstack.com/
[5] http://puppetlabs.com/presentations/keynote-puppet-scale-%E2%80%93-case-study-paypals-learnings
[6] https://labs.spotify.com/2013/05/17/devops-management/

be considered immature there is a large body of scientific knowledge on the subject compared to the of field of software defined infrastructures. Models, such as the ISO 25010 [16], have been created to define what quality is for software products. Such quality models do not exist yet for determining the quality of software defined infrastructures.

1.2 Research questions

As there is only a small body of scientific knowledge on the subject of software defined infrastructures, this study attempts to create a fundament in quality measurement of software defined infrastructures.

The first research question is formulated as:

RQ1 How can we define quality of a software defined infrastructures?

The first research question can be subdivided into smaller questions:

RQ1.1 What are quality characteristics of software defined infrastructures?

RQ1.2 How can we create a quality framework for software defined infrastructures?

During the identification of quality characteristics of we discovered that testing is an important quality characteristics for software defined infrastructures. We decided to operationalise the quality of testing by to creating a test quality evaluation model for software defined infrastructures.

The second research question is therefore formulated as:

RQ2 How can we evaluate the testing quality of software defined infrastructures?

We can subdivide this research question into:

RQ2.1 What are important aspects for testing in a software defined infrastructure project?

RQ2.2 How can we create a model that is capable of measuring the testing quality of a software defined infrastructure?

RQ2.3 How can we create a model that is accepted by practitioners in the field of software defined infrastructures?

1.3 Research method

1.3.1 Creation of SDI quality framework

We identified important quality aspects of software defined infrastructures by interviewing practitioners on the subject using semi-structured interviews. In these interviews we asked the practitioners what they think is important when trying to maintain a level of quality in their own software defined infrastructure projects. We asked them what they would advise for a project which they are not involved if they were hired to give advice.From these interviews we created an overview of aspects mentioned by each interviewee and used that as input for determining which quality characteristics these aspects stand for.

We chose a qualitative approach approach because of the inductive nature of qualitative research, allowing us to find new aspects new quality aspects, where a deductive approaches would only allow us to test existing ideas. The reason that we chose to do interviews is because practitioners were at hand while literature on the subject of software defined infrastructures was limited.

1.3.2 Adapting the SIG testing quality framework for SDI

The Software Improvement Group (SIG) has a quality model [22] to evaluate the testing quality of software which we used as a baseline testing quality model for software defined infrastructures. As with the creation of the quality model for software defined infrastructures we interviewed practitioners to get input to determine what a testing quality model for software defined infrastructures should encompass.

We applied the SIG testing quality model to two software defined infrastructure projects using semi-structured interview to answer all the questions that the SIG testing quality model consists of. We used the result of the application to determine if the SIG quality model was suitable for software defined infrastructures.

After application of the SIG testing quality model, we thought that the model did not fit software defined infrastructures, but we also thought that we did not have enough information to improve the model. To get the information we needed we interviewed practitioners in the field of software defined infrastructures. We asked the practitioners which themes of the SIG testing quality model they thought were relevant to software defined infrastructures and which themes they thought were irrelevant. In the same interview we asked the practitioners questions related to the acceptance of the current model and what themes, related to software defined infrastructures, they were missing from the current testing quality model.

1.4 Thesis outline

In the second chapter we provide a background on software defined infrastructure to make the topic more tangible. It contains examples of how different parts of software defined infrastructures can be implemented and contains examples of successful implementations in the industry.

The third chapter answers the first research question, that is, which quality characteristics can be identified for software defined infrastructures. We explain in more detail what research method we applied to get the data and to make sense out of the data. And we end the chapter by providing a quality model for software defined infrastructures.

In the fourth chapter we create a testing quality evaluation model for software defined infrastructures. We cover the results of assessing projects using the SIG testing quality model. Then we cover the results of the interviews on the themes and acceptance of the SIG quality model and interpret the results. We end the chapter by providing an improved testing quality model for software defined infrastructures.

We conclude this thesis in the fifth chapter where we will summarise the results, discuss the strengths and weaknesses of our approach and by providing pointers for future work.

Chapter 2

Background

In this chapter we provide background on the concepts that are part of this thesis. We start by explaining what a software defined infrastructures are by giving a definition and by putting it in context with other concepts. The concepts surrounding software defined infrastructure include configuration management tools and languages, cloud computing and the DevOps philosophy. We explain what each of these concepts are and how they are related to software defined infrastructures.

2.1 Software defined infrastructures

In this section we provide background on software defined infrastructures. We start by giving an overview of the definitions of software defined infrastructures that are being used by different parties. From these definitions we will pick one definition of what software defined infrastructures is and we will use that throughout the thesis. We then place software defined infrastructures in context with other concepts which are defined as software. We then explain what software defined infrastructures solves by comparing by sketching the situation as it was before the application of software defined infrastructures.

2.1.1 Definition of software defined infrastructure

When talking about software defined infrastructures, there is no agreed upon definition of what it encompasses. Intel refers to software defined infrastructures as a means of data centre automation[1], where hardware components like network and storage devices are virtualised.

In this thesis we define software defined infrastructures more closely

[1] http://www.intel.com/content/www/us/en/switch-silicon/software-defined-infrastructure-sdi-infographic.html

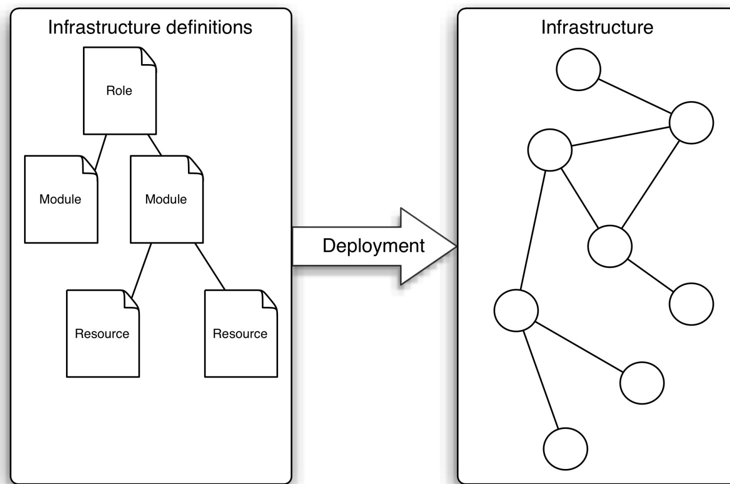

Figure 2.1: Overview of software defined infrastructure

to 'infrastructure as code' as described by IBM[2]. In this sense software defined infrastructures encompasses all the computer code that provisions computers, that is configuring a computer to a desired state so that it performs a certain task, such as serving a website to end users.

Central to this thesis will be code written in configuration management languages such as the Chef language and the Puppet language. Software written in these languages should define what the desired configuration of a computer is. The desired state could be that certain applications should be or should not be installed, that they are configured in a certain way, and that these application should be active or inactive.

Figure 2.1 depicts what software defined infrastructures are on a high level. On the left hand side is a collection of files containing the definitions of what the infrastructure should look like. In the middle is a process of deployment, which takes the definitions as input and turns these into actions that the computers should perform to reach the desired state. On the right hand side is the resulting infrastructure providing services to users from within an organisation or to clients outside of the organisation.

Also included in our concept of software defined infrastructures is all the software written to "create" and "destroy" computers. The creation and destruction of computers mostly plays a role in virtualised environments such as with cloud computing. Creation of a computer means launching a new logical instance of a computer and destruction means turning such an instance off. Automating creation and destruction of computers is often called 'orchestration'[3] and combined with the automatic provisioning of computers

[2]http://www.ibm.com/developerworks/library/a-devops2/

[3]https://www.ibm.com/developerworks/community/blogs/9e696bfa-94af-4f5a-ab50-

leads to a having a high amount of automation of the management of an IT infrastructure.

2.1.2 Other software defineds

Software defined infrastructure is also an umbrella term for other things 'software-defined', such as:

Software-Defined Networking (SDN) A network architecture where network logic is centralised in a software-based controller, which maintain a global view of the network. This allows network administrators to programmatically manage a simplified network instead of having configuration files scattered across among many devices. [12]

Software-Defined Storage (SDS) A storage architecture where administrators and users get functionality in terms of capacity, performance and reliability and are being abstracted from hardware configuration of storage. [15]

Software-Defined Data Centre (SDDC) A term encompassing many 'software defined'-terms. It provides Application Programmable Interfaces (API) for low-level components to administrators of the data centre as well as to tenants of the data centre. [9]

In the context of this thesis, software defined infrastructure is described as software that orchestrates and provisions computers so that they form a computer infrastructure.

2.1.3 The problem software defined infrastructures solve

Infrastructures not defined by software

In an organisation where computers are not provisioned using software defined infrastructures, the worst-case scenario when a computer needs to provisioned would be that the network administrator would have to:

1. Install an operating system using a CD.
2. Configure the operating system.
3. Install the required applications using a CD or package repository.
4. Configure the applications.
5. Run the applications.

c955cca76fd0/entry/orchestrating_the_cloud_to_simplify_and_accelerate_service_delivery1

Provisioning a computer this way can take up to hours, meaning the system administrator is limited in what he can do in the meantime. The fact that this takes a lot of time also means that infrastructure creation is inflexible meaning that it will be hard to deal with rapidly changing environments. This method of provisioning is also error-prone. A human having to perform a certain set of tasks over and over is bound to perform the set of tasks differently each time. In system administration this leads to so-called 'works of arts' where many systems are configured in different ways and only a single administrator knows how a system is exactly configured.

In the best-case scenario when provisioning a computer without using software defined infrastructures, the system administrator has an image of a fully configured computer which he can load onto as many computers as he likes. This however requires that of every type of configuration required by an organisation, the system administrator would have to create an image of it, which by itself is time consuming, but also is inflexible and additional configuration steps would still lead to so-called 'works of art'.

Infrastructures that are defined by software

In an organisation where computers are provisioned using software defined infrastructures, the provisioning of a computer means that a definition of how that computer should be provisioned has to be created. When this is done the steps of provisioning a computer becomes:

1. Install the operating using a CD or image.
2. Install configuration management tool, such as Puppet.
3. Provide definition to the configuration management tool.

The first two steps can be eliminated by making use of infrastructure-as-a-service cloud computing. This means that in a best-case scenario it is possible to fully automate the provisioning of a computer. The result is that it takes less time of a system administrator to provision a computer, meaning that he manage more computers. It also results in that computers doing similar tasks are configured in a homogeneous way, making the overall infrastructure more manageable.

2.2 Infrastructure management using Puppet

At the centre of software defined infrastructures are the configuration management tools and languages such as Chef and Puppet. The configuration management languages are a means of defining what the desired configuration of a computer is. The configuration management tools are a means

to transform these definitions into actions that a computer can perform to reach that desired state.

In this section we provide examples of code written in the Puppet language and we describe how the Puppet tool turns this code into action to be performed by the computer. The examples in this section are based on Puppet but the general concepts are also applicable to other configuration management tools and languages.

2.2.1 The Puppet language

A major difference between Puppet and many traditional programming languages is that Puppet is a declarative language and not an imperative language. An example of an imperative language is the programming language C. To describe in the C programming language what you want a function to do, you need to fully write out all the intermediate steps.

In a declarative language the developer describes the desired result and the compiler or interpreter figures out how to reach the desired result. An example of a declarative language is Structured Query Language (SQL), which is used to query databases. In SQL, the programmer explains that it wants the database to provide data from specific columns given a certain constraint on those columns e.g. “From the table people, provide me the names of people who are of age 18 and older”. When given a query statement, the SQL interpreter will then decide the strategy it will apply to get the desired result.

```
file {'ntp.conf':
  path    => '/etc/ntp.conf',
  ensure  => file,
  content => template('ntp/ntp.conf')
}
```

Listing 2.1: Example of a resource definition.

The declarative nature of Puppet allows the developer to define what kind of resources he wants applied to a computer[4]. Among the default resource types are the types ‘File’, ‘Service’, ‘Package’ and ‘User’. When declaring a resource you can give properties to it. The most prominent properties of a resource are its handle and the ‘ensure’ property. The handle of a resource can be seen as a reference or name of a resource. With the ensure property the developer can tell Puppet that it has to ensure the presence or absence of that resource. For example with the File-resource, you can ensure that a file is present or absent, or more detailed ensuring that

[4]https://docs.puppetlabs.com/references/latest/type.html

the specified path it is a file, a folder or a symbolic link. With the Service-resource Puppet can ensure that the service (An application running the background, such as the system clock) is running or that it is stopped.

An example of a resource definition can be found in listing 2.1. In this example a File-resource is created with the handle 'ntp.conf'. The specific file the resource is referring to is '/etc/ntp.conf' and it is to be ensured that this file exists. The content of the file has is generated using the specified template, which acts as a blueprint for how the content of the file should be structured.

When a resource is provided as input the Puppet, the Puppet tool will try to reach the desired state as defined by the resource. In the File-resource example, each time the Puppet tool is run, it will make sure that the file is there and if it is not, it will create the file. The Puppet tool will also check if the content of the file is right and if it is not, it will insert the correct content. In general, if the configuration in the definition is not the same as the actual configuration (divergence from the desired state has occurred) then Puppet will make the changes to the that are required (convergence towards the desired state) to reach the desired state of the computer.

Classes, modules and node definitions

Resources in Puppet are bundled in classes[5]. Such classes typically represent applications and include the resources needed to install, configure and run that application. It is typically in these classes that developers generalise their code so that it the classes can be applied to different operating systems. An example of where this is useful is for different Linux distributions with different package repositories. Names of packages of applications can differ over repositories. A concrete example is that the Apache web server is contained in the 'apache2' package in Debian package repository while it is contained in the 'httpd' package on Red Hat-based package repositories[6]. Creating specific cases for different distributions allows a class to be portable, which can be valuable when changing cloud infrastructure providers or when you have to provide the same applications for clients using different operating systems.

One level above classes are modules, which are folder structures containing classes, binary files, configuration file templates and unit tests[7]. Modules are meant to be easily re-distributable and are similar to libraries in traditional programming languages. Puppet Labs offers a central repository for these libraries at Puppet Forge[8]. The Puppet Forge contains modules created and shared by Puppet Labs and by the community.

[5] https://docs.puppetlabs.com/puppet/latest/reference/lang_classes.html

[6] https://docs.puppetlabs.com/guides/passenger.html#install-apache-2

[7] https://docs.puppetlabs.com/puppet/latest/reference/modules_fundamentals.html

[8] https://forge.puppetlabs.com/

```
node 'www1.example.com' {
  include common
  include apache
  include squid
}
node 'db1.example.com' {
  include common
  include mysql
}
```

Listing 2.2: Examples of node definitions.

The last part of the Puppet language we discuss are node definitions[9]. In node definition it is describe which set of modules or classes are to be applied to a computer.

Whenever a computer with a Puppet Agent connects to a Puppet Master server, the Puppet Master will have to decide which configuration to apply to the Agent. The node definitions are lists of classes to be applied to the .

Examples of two node definitions can be found in listing 2.2. If an Agent with the host-name 'www1.example.com' connects to the Puppet Master, then the Puppet Master will instruct the Puppet Agent to apply the 'common', 'apache' and 'squid' classes.

The Puppet deployment tooling

In traditional software development, source code is transformed to an executable computer program by providing the source code as input to a compiler or an interpreter. The job of the compiler or interpreter is to translate the source code into machine code that the processor is able to execute.

In case of software defined infrastructures using configuration management tools such as Puppet, the code is not compiled into machine code but instead into a set of configuration steps called the 'catalog' which the agent then performs. Figure 2.2[10] provides an overview of the catalog creation of the Puppet tooling.

The configuration of a computer starts with the Puppet Agent on that computer doing a catalog-request to the Puppet Master. In this catalog-request the Puppet Agent sends its host name name and a set of facts, including hardware information and software information like which operating system and which version of Puppet are running on the computer. Using the host name information the Puppet Master determines whether the computer is eligible to receive a catalog and if so which node definition

[9]https://docs.puppetlabs.com/puppet/latest/reference/lang_node_definitions.html
[10]Figure found at: https://docs.puppetlabs.com/learning/agent_master_basic.html

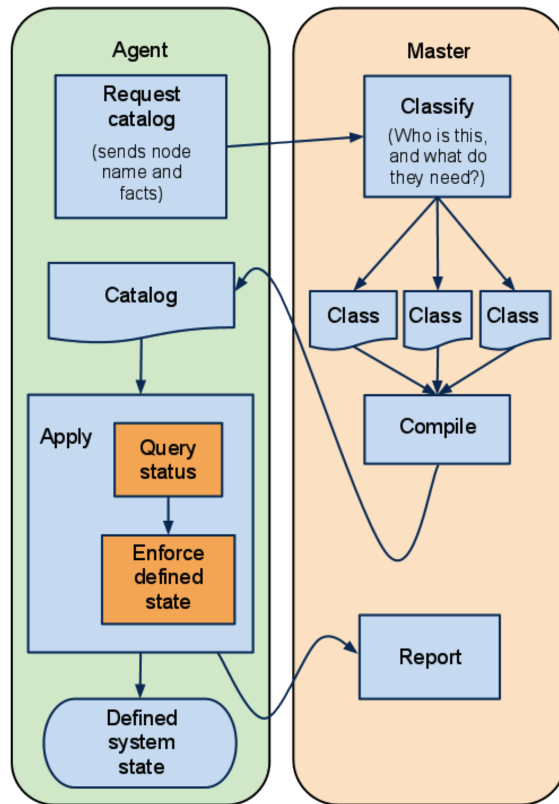

Figure 2.2: Puppet catalog creation and communication

applies. Then, using the facts the Agent has provided, the Master compiles a catalog containing the desired state the machine should converge to and sends that catalog back to the Agent. The Agent then applies the catalog, first by checking if it is already at the desired state. If that is the case then the Agent is finished, if not then the Agent has to take actions to get to the desired state. If all went well, the Agent reports success to the Master, meaning that it has reached the desired state. If things went wrong, the Agent reports failure to the Master, which should alarm the infrastructure developers.

2.3 Cloud computing

While software defined infrastructures are not necessarily a part of cloud computing, the two do go hand in hand when automating IT infrastructures.

The National Institute of Standards and Technology (NIST) defines cloud computing as “a model for for enabling ubiquitous, convenient, on-demand network access to a pool of configurable computing resources that

can be rapidly provisioned and released with minimal management effort or service provider interaction" [18].

The NIST also provides a list of essential characteristics of cloud computing:

- On-demand self-service
- Broad network access
- Resource pooling
- Rapid elasticity
- Measured service

The NIST also distinguishes cloud computing into three service models:

Software as a Service (SaaS) Providing a software application to the end-user, like e-mail or a bookkeeping application.

Platform as a Service (PaaS) Providing a platform to deploy applications on.

Infrastructure as a Service (IaaS) Providing IT infrastructural services, such as servers and storage.

In practice making use of cloud computing means the outsourcing of services provided internally by the organisation to providers external to the organisation. For a small company this could mean that they would not have a mail server inside their organisation but that they would use a SaaS e-mail provider, where the mail service is pooled with the mail service of a lot of other companies.

The term cloud encompasses a sort of vagueness in that you now 'bring' service to the cloud. This means that you outsource a service and that it is practically unknown where this service is physically located.

Virtualisation is one of the foundations of cloud computing, meaning that the hardware is abstracted away. Whenever you make use of a service, such as getting a virtualised servers the server is logical and not physical. A logical server means that it is not bound to a physical machine, meaning that when a cloud provider provides such a virtual machine, it will probably reside on different physical machines each time the server is provided.

2.3.1 Relation with SDI

The relation of cloud computing with software defined infrastructures is that they go well together automating whole IT infrastructures. One can use IaaS providers for generating virtual machines which can then be provisioned using configuration management tools. Many IaaS providers have so-called

Application Programmable Interfaces (API) for the creation and destruction of virtual machines. These API's mean that you can automate the creation and destruction, and using configuration management tools allow automating the provisioning. With a minimum amount of human interaction one can now thus create IT infrastructures. When done well, this means that you can have elastic infrastructure, which grows when there is a high demand and shrinks when there is a low demand for services.

2.4 DevOps

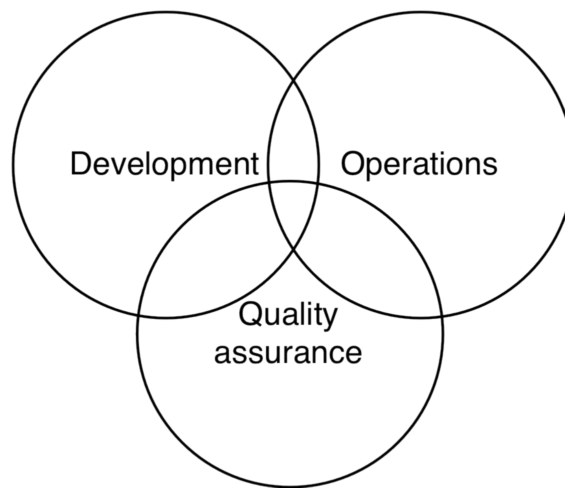

Figure 2.3: Diagram portraying DevOps

In many organisations development and operations are separate parts of the organisation[11][12]. While both are part of a chain with the goal to deliver IT services, both roles have conflicting goals. On one hand development, with the increased adoption of agile development methodologies, have the goal to make changes to software so that the organisation can meet its goals. On the other hand operations, are tasked with providing a stable IT infrastructures so that the continuation of the business is guaranteed. This conflict of goals result in a lack of alignment causing release cycles to take longer than needed.

To DevOps philosophy tries to close the gap between development and operations. By applying development practices to IT infrastructure operations, the deployment cycle time for software releases is brought down. Focus with the DevOps lies with the unification of processes, such as agile and ITIL, on the unification of tooling, such as the use of configuration man-

[11] http://dev2ops.org/2010/02/what-is-devops/

[12] http://www.jedi.be/blog/2010/02/12/what-is-this-devops-thing-anyway/

agement languages and continuous integration platforms, and to improve the communication between the different IT parts of an organisation. Figure 2.3 portrays the desired result of applying the DevOps culture within an organisation. Different IT departments within organisation should collaborate rather than have a conflicting goals.

The relation between software defined infrastructures and DevOps lies in the unification of tooling. With software defined infrastructures, the creation of infrastructures becomes part both of development and operations. Developers can now create and test their software on computers that have for which code has been created using configuration management languages. Using continuous integration tools such as Jenkins[13], each change to the code can be tested automatically and deployed once the tests succeed. With infrastructure becoming code, this now also applies to infrastructures themselves.

[13]http://jenkins-ci.org/

Chapter 3

Identification of quality aspects

In this chapter we create a quality model for software defined infrastructures. We start by giving some background information on quality models for traditional software systems. We then explain how we interviewed practitioners of software defined infrastructures to gain information on quality aspects in that context. We then interpret the results and end by creating a quality model for software defined infrastructures.

3.1 Background

In this section we describe the small body of scientific knowledge on software defined infrastructures. The industry however, has already adopted software defined infrastructures and many practitioners in the field are writing articles about it in the form of blogs. We provide an insight in some of these articles written by practitioners to show the current state of adoption of the software defined infrastructures. We end the background by providing how quality is measured in traditional software engineering.

3.1.1 Small body of scientific knowledge

With the rise of configuration management languages, cloud computing and DevOps there have been many companies adopting software defined infrastructures to make their IT infrastructures more manageable. The increasing adoption of the technology by the industry is not reflected in academic literature as there is almost no literature on the subjects of software defined infrastructure or infrastructure as code.

A model based testing approach for testing idempotence of infrastructure as code operations was proposed by [14]. Idempotence of infrastructure operations means that tasks, such as those generated by configuration man-

agement language tools, are repeatable. The article shows that, using the testing approach, many Chef cookbooks maintained by the Opscode community[1] contain tasks found to be non-idempotent.

The author of CFEngine[2] configuration management tool has written articles on the formalisation of system administration processes. Burgess created a theoretical framework for system administration [8], in which the notion of policy is defined. In [7], Burgess defined idempotence and convergence in the context of system administration are defined and for both concepts a theoretical framework is created.

There are many blogs about increasing the quality of software written in the Chef and Puppet languages. These blogs focus on the usage of testing platforms and the usage of code style checking[3][4][5], but are lacking to define what quality in software defined infrastructure is.

This, to our knowledge, is the current state of research in the field of software defined infrastructures, infrastructures as code and DevOps. Because there is no definition of quality on software defined infrastructure, we aim to make a foundation by creating the notion of quality in this thesis. As a software defined infrastructure is code, we will focus at looking at it from a software engineering perspective. This is because software engineering, which can be seen as an immature engineering principle when compared to traditional engineering principles[6], is in our opinion, a more mature field of research and standardisation than IT-infrastructure engineering is.

3.1.2 Quality aspects of traditional software

Many quality models have been written on traditional software. We provide an overview of some well known software quality models.

Boehm's software quality model

Boehm et al. [5] defines software quality in terms of portability, reliability, efficiency and maintainability. The quality characteristics are decomposed into sub-characteristics if it is implied that an increase in quality of a sub-characteristic means an increase in quality of the supra-characteristic. The article does not go into great detail on how the quality characteristics were determined and how metrics for these quality characteristics were made.

[1] http://community.opscode.com/cookbooks/
[2] http://www.cfengine.com/
[3] http://www.neverstopbuilding.com/foodcritic/
[4] https://www.paydici.com/blog/increasing-chef-code-quality/
[5] http://puppetlabs.com/blog/software-quality-and-your-devops-tool-chain/

McCall's software quality model

McCall et al. [17] determines 55 software quality factors based on a literature study. To make the list of quality factors more manageable the quality factors were then grouped by removing factors that are synonymous and then by logically grouping the factors. The result is the definition of eleven factors and many sub-factors.

ISO 25010 software quality model

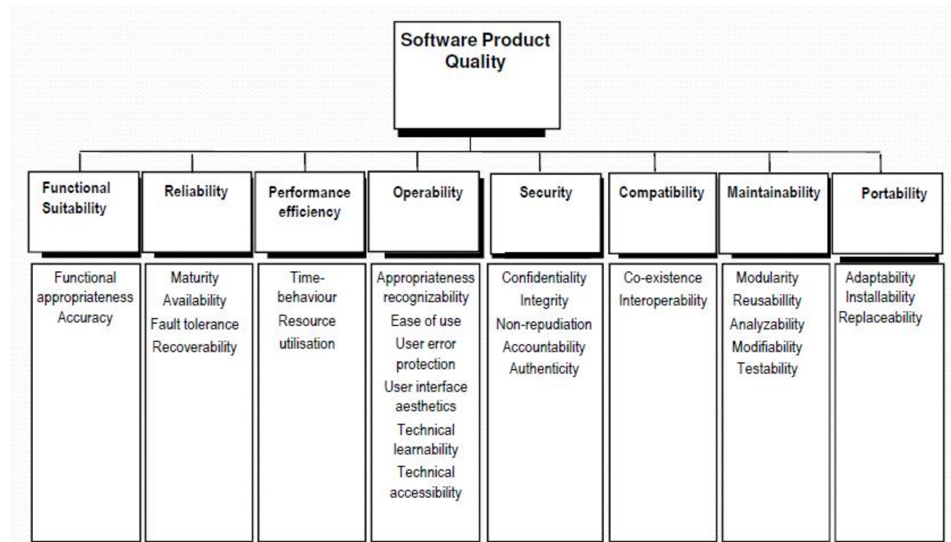

Figure 3.1: Product quality aspects defined in ISO 25010

The International Organisation of Standardisation (ISO) is an organisation that has the creation of standards as its primary goal. The standards created by the ISO vary in field and theme. For almost any kind of field or subject there is a standard describing that thing or process. Examples of standards are ISO 22000[6] for food safety management and ISO 27001[7] on the process of creating a information security management system.

The ISO 25010 standard on systems and software quality requirements and evaluation defines a set product quality characteristics and quality in-use characteristics for systems and software [16]. Product quality characteristics are properties of the system that can be evaluated by looking at the product as-is. An example of a product quality characteristic that the standard defines is 'interoperability', which is defined as: Degree to which two or more systems, products or components can exchange information and

[6]http://www.iso.org/iso/home/standards/management-standards/iso22000.htm
[7]http://www.iso.org/iso/home/standards/management-standards/iso27001.htm

use the information that has been exchanged. Next to the product quality characteristics are the quality in-use characteristics, these in-use characteristics are properties that depend on the context in which the system is being placed. An example of such a property is 'effectiveness', which is defined as: Accuracy and completeness with which users achieve specified goals.

The 25010 standard is the replacement for the ISO 9126[8] standard on software engineering product quality. The 9126 standard supports the needs to assess software quality to ensure value to stakeholders. By creating a set of quality requirements, the technical commission creating the standard intends to provide a set of characteristics so that they can be used to for creating requirements, the criteria for satisfaction and measurements. The replacement added quality in-use characteristics and extended the scope to include computer systems instead of just software products. Other changes are the inclusion of several characteristics and renaming some for more accuracy.

Overlap in quality models

Al-Qutaish compared different software quality models [1]. The conclusion of the article is that the models show a great deal of overlap in the quality characteristics they define. The quality models compared usually differ in determining which characteristics are major characteristics and which ones are sub-characteristics. It is concluded that the ISO 9126 quality model is the most useful one as it is based on consensus.

3.2 Gathering SDI quality aspects

In this section we explain how we gathered quality aspects of software defined infrastructures. We start by explaining the research method we applied to gather information, which is interviewing. We then describe how we transcribed these interviews and how we extracted quality aspects from them. We end this section by giving an overview of the results of the interviews, namely a list of quality aspects the practitioners of software defined infrastructures found important.

3.2.1 Research method

We aim at finding which quality aspects are important in a software defined infrastructure context. Due to the explorative nature of finding new quality aspects, we chose a qualitative research approach. Qualitative approaches allow us to gather new information which can later be tested using quantitative approaches.

[8]http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749

For gathering quality aspects of software defined infrastructures, we chose to interview practitioners of software defined infrastructures using semi-structured interviews. Semi-structured interviewing allows the interviewer to explore subjects as they come up but also to guide the subjects using a pre-defined interview guide. Given their background as consultants or at least having worked for a consultancy firm, we expected the interviewees to be able to reason about software projects outside of their own projects. The interview guide can be found in section 3.2.1.

We have diversified our pool of interviewees by getting interviewees with different backgrounds. Four out of five interviewees were employees of the Software Improvement Group (SIG). These four interviewees were all part of a team responsible for creating the internal software defined infrastructure. Two out of the four interviewees were technical consultants with a development background. The other two interviewees have are part of the IT operations of SIG. Out of these two one has a development background and one has a systems administration background.

The Software Improvement Group describes itself as: "SIG is committed to delivering practical, actionable insights that enable our clients to reduce their software costs, increase their software systems effectiveness and shorten IT project delivery times.

We provide objective, actionable advice on IT landscapes and systems, and guide the development team in achieving optimal results. Our independence and impartiality are important to our clients, who also benefit from our partnerships with leading authorities, such as TÜViT and universities worldwide."[9].

The fifth interviewee is an infrastructure engineer at Schuberg Philis. He has a background in software development and in research at the Software Improvement Group.

Schuberg Philis describes itself as: "Schuberg Philis is an innovative business technology company. We focus on the mission critical applications that our customers and society rely on 24/7."[10]

Interview guide

The interview guide applied for data to determine quality aspects of software defined infrastructures is:

1. Internal project
   (a) What are important quality aspects you look out on your internal project?
   (b) What do you do to keep the quality of your project high?

[9] http://www.sig.eu/en/About_SIG/
[10] https://www.schubergphilis.com/

2. If you were to act as an adviser on an external software defined infrastructure project, what are quality aspects you would look out for and what would you advice?

Transcribing

To save time, transcriptions are not word-for-word written versions of the interview but are summarised reports of the interviews where diverges from the main theme are omitted if they were not contributing to the research goal. The reports of the interview can be found in appendix A. The recordings of the interviews have been handed over to the academic supervisor of this thesis.

We extracted quality aspects of software defined infrastructures by coding the transcriptions based on themes that were identified by during and after the interviews. Appendix B shows the result of the coding, pointing out what the interviewees said about a certain topics.

3.2.2 Results

Overview of the quality aspects

Table 3.1 gives on overview of quality aspects mentioned during the interviews with the practitioners of quality characteristics of software defined infrastructures. The check marks indicate that the particular quality aspect has been mentioned during that particular interview as being something that is important to look out for when looking at the quality of a software defined infrastructure. The last column shows the amount of interviews the particular aspect was mentioned in, it is assumed that more important topics are mentioned in more interviews than topics that are less important.

Interpretation of the interviews

In this section we interpret the results of the interview as shown in table 3.1. We highlight the characteristics that were mentioned in most interviews. Criteria for highlighting was arbitrarily chosen to be characteristics that were mentioned in three or more interviews. During the highlights we mention why the interview subjects feel these characteristics are important. Transcriptions of the interviews can be found at appendix A.

Automation Level of automation or autonomy is a quality characteristic that was mentioned by three interviewees.

Interviewee #2 pointed out that the amount of manual operations performed on servers is something to look out for.

Interviewee #3 and interviewee #5 explained that automation is a characteristics that tells something about the predictability and maturity of a

	Interview					
Characteristics	**1**	**2**	**3**	**4**	**5**	**#**
Audit trail				✓		1
Automation		✓		✓	✓	3
Best practices		✓	✓	✓		3
Co-existence	✓					1
Complexity					✓	1
Data separation			✓			1
DevOps		✓				1
Environments	✓				✓	2
Extensibility					✓	1
Functional suitability					✓	1
Infra design				✓	✓	2
Level of freedom					✓	1
Maintainability		✓	✓		✓	3
Maturity					✓	1
Monitoring		✓			✓	2
Node cohesion	✓					1
Predictability				✓		1
Test automation	✓	✓				2
Testability		✓				1
Testing	✓	✓	✓	✓	✓	5
Testing effort				✓		1
Tools usage	✓	✓				2
Version control	✓	✓				2

Table 3.1: Overview of quality characteristics mentioned during interviews.

software defined infrastructure. The predictability is increased when automating the deployment using a deployment pipeline which includes automating unit testing and integration testing. Other areas that were mentioned as candidates for automation include incident handling of the infrastructure, meaning that a monitoring system checks the infrastructure for incidents and that incidents caught by the monitoring systems are automatically handled. The usage of cloud orchestration, which is tooling that manages the infrastructure by creating and destroying nodes and by provisioning the nodes, was also mentioned as something that could be automated.

Best practices Three subjects mentioned the application of best practices.

Interviewee #2 mentioned that the tooling available for software defined infrastructures does less for the developer than in many other programming languages. This means that the developer should employ best practices more

as they are a set of 'proven' practices making less room for error.

Interviewee #3 explained that the configuration management languages are owned by a single companies and are therefore subject to big changes. This makes it important to closely watch the blogs of the companies which are in control of the languages as those are important channels of communicating these changes.

Interviewee #3 and interviewee #4 pointed out language specific code smells of the Puppet language and the best practices would be to not introduce those smells in your code. The first code smell that was mentioned is programming in an imperative style in the Puppet language as the Puppet language is a declarative language. The declarative nature means you can tell the Puppet Master what you want a machine to look like and the Puppet Master will figure out the steps to get there. Using an imperative approach – such as by using require statements, explicitly telling the Puppet Master in which order to do things, thus creating dependencies – overrules the order in which Puppet executes commands which could lead to circular dependencies. Another code smell mentioned is overusing of the exec-statement. The exec statement of the Puppet language means that the code a command is executed directly to the command-line interface. The Puppet Master has no idea of what happens on the system when executing these external commands. The Puppet Master will thus be unable to tell where to schedule these exec statements and will be unable to tell if the exec statement succeeded or failed, making it hard to detect errors generated by the external commands.

Maintainability Interviewee #2 and interviewee #3 explained that they would look at certain static code indicators of maintainability. The indicators they mentioned were unit length, parameter list length, good naming practices and code complexity. Interviewee #2 said that the McCabe measure for complexity is not that useful for code written in the Puppet language due to its nature as a domain specific language in which programmers do not have the tendency to create a lot of nesting. Code volume would therefore be a better measure of complexity. Interviewee #5 mentioned that he uses Sonar Qube[11] to do static analysis of his code base.

The method of keeping the code maintainable applied by the team of interviewee #5 is by continuously refactoring on the code base. This means that, in line extreme programming and agile principles, developers are allowed to refactor code whenever they see it fit. Another thing they the team does to keep the code maintainable is by being forced to test everything. This creation of tests forces developers to go back and forth between creating tests and writing testable code. Creating more testable code means also that the code is more maintainable[13].

[11] http://www.sonarqube.org/

Testing All interviewees mentioned that testing is something they find important in their own project and test automation was also mentioned as something they would look out for.

There are however some disagreeing views on the importance of unit testing. Interviewee #2 and interviewee #3 mention that unit testing Puppet code is not as important as integration testing as unit tests is simply code written a different language for what you already wrote in the Puppet language. Combined with the shared opinion of interviewee #1, that the provisioning of computers is fragile due to a large amount of external dependencies such as the operating system, they feel that integration testing is far more important than unit testing. Interviewee #5 thinks that unit testing is very important as it is a way to test to test units which are generalised to work with more than one operating system.

The importance of integration testing is shared by everyone. The interviewees mentioned that integration testing is important as deploying an infrastructure and then testing against that infrastructure is a good way to test if the code actually deploys the infrastructure as it is supposed to.

Interviewee #1, interviewee #2 and interviewee #3 explained how their team tests the whole infrastructure on a daily basis. By rebuilding the infrastructure in the staging environment on a daily basis, they test if their infrastructure is able to reach the desired state from the ground up and is not only able to transition from state to state.

3.3 Construction the SDI quality model

In this section we transform the results of the interview into a quality model tailored to software defined infrastructures. We start by grouping the quality aspects into logical, SDI related, groups. We then make a visual representation of the model and explain it.

3.3.1 Grouping the characteristics

The goal of the model is to measure the quality of the software defined infrastructure. In chapter 2 we explained that software defined infrastructures consist of the software, the deployment process and the resulting infrastructure. We can thus decompose the quality of the software defined infrastructure into the quality of each of those parts. This makes the quality of software defined infrastructures the aggregation of:

- The quality of software
- The quality of the deployment process
- The quality of the infrastructure

For each of the parts of the software defined infrastructure we determine quality characteristics based on the aspects given as important by the interviewees.

Quality of the software

Interviewees mentioned that the usage of best practices, proper tooling and version control is important. Other than these aspects, interviewees mentioned code metrics related to code maintainability. We combined these aspects into a maintainability quality characteristic for the quality of software of the software defined infrastructure. The application of best practices and usage of proper tooling allow the developers to keep to code maintainable in an field where there is a lack of good tooling. Version control is an important maintainability aspect as it allows a team of software engineers to collaborate on a code base and revert to prior versions of the software in case changes have broken the software. Code metrics in a static code analysis can provide a good tool for measuring the maintainability of the software.

Software defined infrastructures can be used to create portable infrastructures. Portability can be split up in two aspects. The first aspect is the ability to deploy an infrastructure independent of the operating system installed on the computers in the infrastructure. This is helpful in the case of when it is decided that the operating systems of the computers should be changed due to performance or security reasons.

The second aspect is the ability to be able to switch cloud service providers with little effort. Reasons for wanting to change cloud service providers could be cost related or for legal reasons. Costs are a concern due to competition on the cloud market, where some cloud service providers try to do lock in users[2]. Legal issues are a concern as government agencies are gathering more and more data from IT companies[12][13]. This results in companies moving data from countries foreign to their customers to the countries their clients are located in, so that the data of a client does not fall under the jurisdiction of a country which is foreign to the customer.

Quality of the deployment process

Quality aspects mentioned by the interviewees related to the deployment process are deployment automation using a build pipeline, testing code before moving it into production and the testability of the infrastructure. The

[12] http://www.reuters.com/article/2014/07/31/us-usa-tech-warrants-idUSKBN0G024I20140731

[13] http://www.washingtonpost.com/business/technology/tech-companies-urge-lawmakers-to-reform-nsa-programs/2013/10/31/f100ced6-4264-11e3-a751-f032898f2dbc_story.html

inclusion of a build pipeline in the deployment process is a form of deployment automation in which testing can be included. The benefit of a build pipeline is increased automation of deployment, leading to less man-hours spent on deployment and a more predictable deployment process, causing less room for human-made errors.

The quality of testing during the deployment process increases the safety against faults. Preferably every change needs to be automatically tested before the code can be placed in production.

The last aspect is the cycle time of testing and the cycle time of the deployment of the infrastructure. Reduction of the time it takes to test and deploy the infrastructure leads to an increased possibility to fix bugs and changes. It is sometimes decided that an infrastructure can not be deployed whenever there are failing tests, if this is the case then the cycle time of a whole development iteration is limited by the test cycle time.

Quality of the infrastructure

The quality aspects related to the infrastructure that were mentioned were autonomy, recoverability and suitability of the infrastructure.

Recoverability is the ability of an infrastructure to recover from failures or incidents. Monitoring systems should warn system administrators when necessary, but in the first place the infrastructure should be able to repair itself if possible.

Autonomy is closely related to recoverability in the sense that system administrators ideally have to put in as little time as possible maintaining the infrastructure, which decreases the amount of system administrators needed to maintain the infrastructure. A greater autonomy also increases the predictability of an infrastructure as it is less like that human made errors slip in during maintenance activities.

Suitability can be subdivided in technical and functional suitability. Technical suitability means that the infrastructure should be able to handle different kinds of strain. Functional suitability means that the infrastructure must provide the functionality it should, meaning the specific applications the infrastructure provides.

A characteristic related to the technical and functional suitability of the infrastructure is that of complexity. A more complex infrastructure requires more system administrators to maintain it. Either because the complexity require more administrators to maintain smaller amount of computers. Or that the size of the infrastructure requires a large amount of administrators. Example of complexity in an infrastructure are the amount of computers, and the cohesion and coupling between computers.

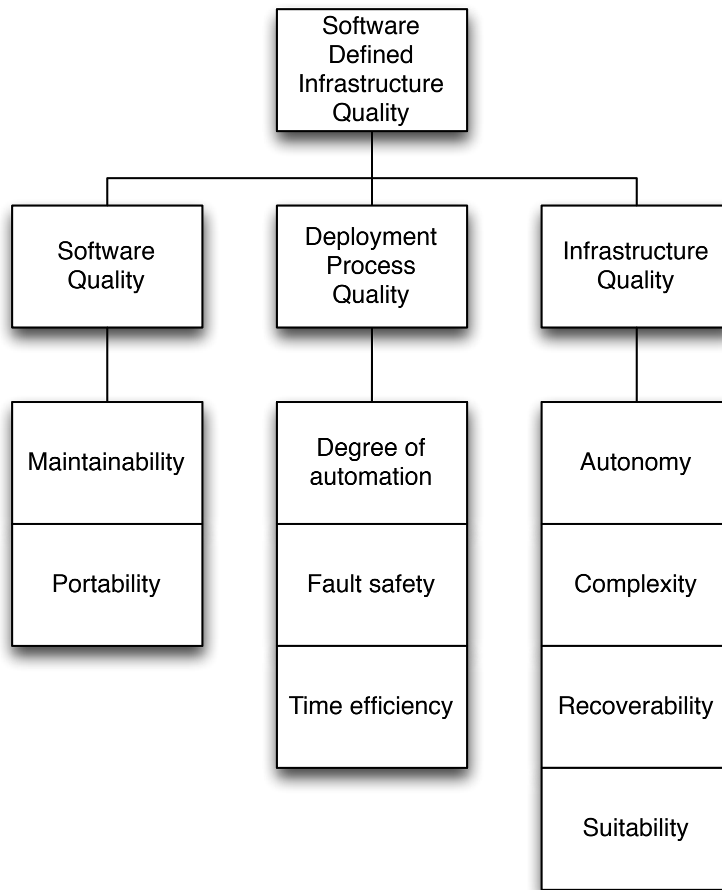

Figure 3.2: Quality model for software defined infrastructures

3.3.2 Visual representation of model

Following the categorisation of the quality aspects mentioned in the interviews into quality characteristics, the model shown in figure 3.2 has been constructed.

The base node is the overall quality of the software defined infrastructure. Each of the decomposed parts of the software defined infrastructure has its own quality characteristics, as shown in the grouping of the aspects. It should be noted that the characteristics of the deployment process and the infrastructure are for most part embedded in the code. Though it is more probable that the quality of these parts are measured in-use rather than by looking at the code.

Characteristics of the quality of code are maintainability and portability:

Maintainability The degree developers are able to make changes to code.

Portability The ease of moving the infrastructure to different hardware, operating systems or cloud service providers.

Characteristics of the quality of deployment are autonomy, fault safety and time efficiency:

Autonomy The degree of automation in the deployment process.

Fault safety The ability of the deployment process to prevent errors.

Time efficiency The time required for deploying the infrastructure.

Characteristics of the quality of the infrastructure are autonomy, complexity, recoverability and suitability:

Autonomy The degree of automation of the infrastructure in place.

Complexity The manageability of the infrastructure.

Recoverability The ability to cope with faults happening inside the infrastructure in place.

Suitability The degree that the infrastructure suits the wishes of the principals.

3.4 Conclusion

In this chapter we presented the results of the interviews we had with practitioners of software defined infrastructures on quality aspects of software defined infrastructures. We then created a quality model for software defined infrastructures containing quality aspects one could look at when assessing the quality of software defined infrastructures.

The next step would be to make a concrete implementation of the quality characteristics by creating metrics to measure the characteristics. In chapter 4 we create a model for measuring the quality of testing in software defined infrastructures. Test quality is one of the pointers for fault safety in the deployment process.

Chapter 4

Testing quality of software defined infrastructures

In the process of interviewing experts on the quality of software defined infrastructure, the aspect that stood out the most as being important was testing. For this reason we decided that it would be helpful to create a quality model for testing in a software defined infrastructure context. In this chapter we provide context on testing and the quality of it of traditional software. We explain the differences between testing in a traditional setting and testing in a software defined infrastructure setting. Then we describe our research method for the creation of a test quality model for software defined infrastructures. We provide the results of the data acquisition and how it has lead to a new model. We end by validating the new model by applying it on the data we gathered to the new model.

4.1 Background

In this section we provide background on why testing is important, how testing is being done and what aspects are to be taken into account when looking at the quality of the testing process. We then compare between testing in a traditional context to testing in a software defined infrastructure context to explain which practices are similar and which practices are different in the two contexts.

4.1.1 Testing as an important quality aspect

When interviewing experts on what they thought were important quality aspects of software defined infrastructures, as described in chapter 3, without exception they included testing as an important aspect. Many of the interviewees even thought that it was the most important quality aspect of software defined infrastructure projects. As we were aiming to instantiate

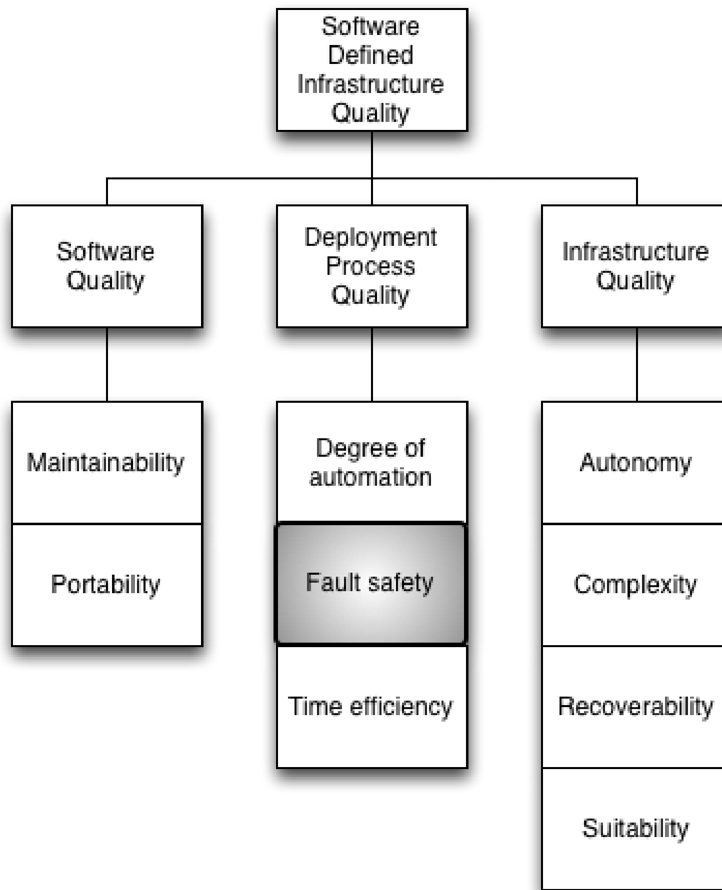

Figure 4.1: Testing as part of fault safety in the deployment process quality of software defined infrastructure quality

one of the quality characteristics of software defined infrastructures, we felt that it would be the most useful to create a testing quality model as a metric for fault safety in the deployment process of software defined infrastructures. Fault safety in the quality model for software defined infrastructures created section 3.3.2 is highlighted in figure 4.1.

The experts on software defined infrastructures explained that testing was very important in this context for various reasons. The most obvious reason why testing is important is related to the DevOps philosophy. Software defined infrastructures is a way of creating computer infrastructures, which prior to DevOps becoming spread was the domain of 'Operations', which is generally the system administrators tasked with creating an infrastructure that has to be as stable as possible. To create a stable infrastructure, system administrators are reluctant to changes in the infrastructure

as each change has the possibility to introduce risks to stability. On the other hand, the trend in application development is to use Agile methodologies, which includes short release iterations. The short release iterations that developers like to have is in conflict with the tendency of operations to create a stable environment by deterring changes. DevOps is the philosophy that tries to solve this conflict by applying software engineering practices on operations. One of these practices is the application of testing in infrastructure management. With the introduction of testing, developers are able to release more often as they guarantee that changes will not break the infrastructure by testing.

4.1.2 The goal of testing

The International Software Testing Qualifications Board (ISTQB) is an organisation that certifies software testers. The ISTQB defines the goals of testing as[19]:

- Finding defects.
- Gaining confidence about the level of quality.
- Providing information for decision making.
- Preventing defects.

The goals of the ISTQB are compatible with [4], where it is described that testing is done to check if something behaves as intended, to identify potential malfunctions and is being used in the industry for quality assurance. While – as these goals point out – testing is important to find defects and to increase confidence in the software, it should be pointed out that testing cannot assure the absence of defects. This general truth was made famous by Dijkstra's: "Program testing can be used to show the presence of bugs, but never to show their absence!" [11].

4.1.3 Testing process

According to [19], the software testing process consists of the generic steps:

Planning and control Defining the objectives of testing and specification of test activities in order to meet the objectives and mission.

Analysis and design Transformation of test objectives into tangible test conditions and test cases.

Implementation and execution Specification of test procedures and test scripts and running them.

Evaluating exit criteria and reporting Evaluation of text execution against the criteria to stop testing and reporting the results to the stakeholders.

Test closure activities Collection of test data to consolidate experience.

The generic formulation of the testing approach by ISQTB is likely to generalise the process so that it can encompass different types of software development methodologies such as the agile and the waterfall methodologies. Using the waterfall methodology, testing is a separate phase in the development of a software product. The testing phase comes after the development phase meaning that testing starts when development of a version of a product is done. In agile methodologies, testing is a part of development meaning that each feature is done only if it is tested to a certain extent.

Making these steps more concrete a software team using an Agile approach could be planning the testing by communicating about the tooling the team is going to use. In a definition of done a team would then describe when a requirement is done, which should include the extent to which a feature is tested. Then depending on the approach tests are written either before or after implementation of the feature or change and is executed. After execution the results of the tests are presented to the developer. In case there are failing tests, the developer will then try to find the root cause of the failing test and will then attempt to fix the bug until the tests are passing.

4.1.4 Testing methods

There are numerous ways of testing a piece of software. The ISTQB foundation level syllabus [19] defines among many things, types of testing and levels of testing.

Types of testing Types of testing are focussed on test objectives, such as functional testing and non-functional testing. Functional testing is described as testing what the system should do. Input for functional tests can be documented specifications or the understanding of the tester about what the system should do.

Another type of testing is non-functional testing, which include any type of testing that is not testing the functionality. The ISTQB foundation level syllabus refers to the software quality model of ISO 9126 for things to test such as maintainability, performance and reliability.

Levels of testing The ISTQB distinguishes four levels of testing:

Component testing Testing the smallest components, such as modules and classes and test tests these components in isolation to test functional and non-functional aspects. Quality of component test code can be measured and is positively correlated with some aspects of issue handling [3].

Integration testing Testing of interfaces between components such as interaction between the components and the operating system or the database. Due to its larger scope its harder to detect the root cause of a failing test than it is with component testing.

Systems testing Testing of the whole system. The goal is to see if requirements are met and if specific use cases can be performed.

Acceptance testing Testing of the software system by the customer or other stakeholders to see if the software meets the (documented) wishes.

4.1.5 Testing in SDI

The types of testing and levels of testing also apply to software defined infrastructures.

Types of testing The functionality of a software defined infrastructure is the 'what' the infrastructure is supposed to do. Testing the functionality of a software defined infrastructure thus means that you test if the modules, systems or infrastructure as a whole does what it is supposed to do.

The non-functional aspects of a software defined can be the same as for a tradition software system. Non-functional aspects can include performance, reliability and security for example.

Levels of testing The units of a configuration management language are typically modules responsible for the management of a single application. An example of testing such a module is testing the conditional branches for generalising the modules to work with different operating systems[1].

Integration testing in software defined infrastructure projects includes testing modules, either in separation or in combination with other modules, while they are are being run against actual machines. An example of integration testing could be the application of a module that installs and runs an NTP client on a machine and then test on the machine if the NTP client is actually running and has been configured[2].

System and acceptance testing in a software defined infrastructure context usually means testing if the applications are accessible from the outside.

[1]http://puppetlabs.com/blog/the-next-generation-of-puppet-module-testing/

[2]http://ehaselwanter.com/en/blog/2014/06/03/integration-testing-infrastructure-as-code-with-chef-puppet-and-kitchenci/

In the case of a web application it means testing if the web application actually loads when browsing to it. The difference between system testing and acceptance testing is that system testing is done by the tester or is done automatically, while acceptance testing is done by stakeholders of the infrastructure.

4.2 SIG testing quality model

The Software Improvement Group (SIG) uses their internally created testing quality model [22] to assess the quality of the testing process of traditional software projects. The testing quality model is confidential and is based for most part on the foundation level syllabus of the ISTQB [19].

In the testing quality model of SIG, four quality characteristics of software testing are defined:

Representativeness The ability of testing to reflect the usage of the software in the production environment.

Effectiveness The ability of testing to detect the cause of a defect.

Efficiency The extent to which time and effort are are optimised for testing.

Evolvability The ability of testing to be adapted to product changes over time.

As one can not measure these characteristics directly, the testing model includes a set of properties which can be measured. These testing properties are mapped to the testing quality characteristics if it can be argued that the property is an indicator for the characteristic.

The testing properties defined by the SIG testing quality model are:

Test scope The level at which the objective and the strategy of testing are defined.

Test validation The extent to which the requirements and the testing are valid during the whole product life cycle.

Test design The extent to which the tests resemble the real world situation in production.

Test environment The extent to which the test environment resembles production and can adapt to changes.

Feedback duration The average duration of a feedback loop from the moment the testing is requested to the moment the test result is given back to the developers.

Autonomy The extent to which the testing can be run independently, assessed by how much man-effort the testing execution requires.

Reporting The extent to which the reporting is informative and actionable.

The mapping of the testing properties in the testing quality model is summarised in figure 4.2 where the testing quality properties are on the horizontal axis and the testing quality characteristics are on the vertical axis.

	Test scope	Test validation	Test design	Test environment	Feedback duration	Autonomy	Reporting
Representativeness	X	X	X	X			X
Effectiveness	X		X				X
Efficiency					X	X	
Evolvability		X		X		X	

Figure 4.2: Mapping of test quality properties to test quality characteristics

The application of the testing quality model is done by (technical) consultants of the software improvement group. Each property is comprised of a set of questions the consultants have to answer. The questions can be answered by interviewing stakeholders, reviewing the artefacts or by on-site observation.

4.2.1 Questions about the SIG testing model and SDI

Our initial thought on the testing quality model of SIG was that such a model would be helpful for assessing the quality of the testing process in software defined infrastructures. We do question the generalisability of the model, meaning that we think that the model might not generalise from traditional software to software defined infrastructures. A lack of generalisability could mean that the model could not accurately measure the quality of testing practices in a software defined infrastructure context. Our idea was that in some areas the model was fitted to cater more towards tradition software development projects using a waterfall-like methodologies.

4.2.2 Finding limitations of the model by application

To confirm the idea that the model was not completely fitting, we applied the testing quality model to two software defined infrastructure projects. The first project we applied the test quality model to was the internal software

defined infrastructure project of SIG. The goal of this project is to move the internal infrastructure to the cloud.

The second project we applied the model on is one from Schuberg Philis. The goal of this software defined infrastructure project is to provide an infrastructure for front-end testing of web applications. The infrastructure has to create machines with different operating system and browser configurations onto which Selenium can be run. Selenium[3] is a front-end testing framework that is can automatically test the front end of a web application. With this project, clients can ask to test their web application on a large array of browser and operating system combinations.

The application of the model was done by interviewing one of the testers of each of the projects. It should be noted that in both these projects agile and DevOps methodologies were applied, meaning in both these cases that developers are also responsible for the quality assurance as these roles are not segregated in the DevOps philosophy. The interview guide for the application of the test quality model can be found in appendix C. The interview was held in a semi-structured manner, meaning that we could diverge from the interview guide if interesting topics had arisen.

The testing quality model has decision trees where questions are asked in a certain sequence, where to get a positive rating for a certain questions, all questions prior to that question must have been answered positively. For example, if a property consists of ten questions and nine are answered positively, but the negatively answered questions was the third. Then the rating of that property would be three out of ten. While in a real world scenario it would be redundant to ask questions from sequences that are already closed, we asked them anyway as they could provide input about the usefulness of rating a property sequentially.

Result of application

The result of the application of the testing quality model for the two projects can be found in table 4.1. The ratings are given for each of the quality characteristics and are aggregated to a final rating by averaging the ratings of the quality characteristics.

Project of Schuberg Philis The final rating of the project of Schuberg Philis is 3.9, which is the aggregation of the four testing quality characteristic ratings as shown in table 4.1.

The testing quality characteristics ratings are an aggregation of the test quality property rating as summarised in figure 4.2. We explain the scores for each of the testing properties.

[3]http://www.seleniumhq.org/

Characteristic	Project of Schuberg	Project of SIG
Representativeness	3.2	2.8
Effectiveness	3.7	3.3
Efficiency	5.0	5.0
Evolvability	3.7	3.0
Final rating	3.9	3.5

Table 4.1: Result of application of the testing quality model on two projects

The project scored a rating of 3 on the test scope property, this is due to a lack of prioritisation of the testing effort. According to the testing model, a good practice would be to prioritise the testing based on how much effort. This prioritisation is needed as it is impossible to fully test everything in a project.

The test validation property gets a rating of 2 as there is are no formalised change requests. The model states that it would be a good practice to formalise change requests as that could be a trigger for changing the tests. In this project however, an agile development methodology is used. When using such a methodology change requesting is not a formal process as product owners generally communicate their wishes directly with the developers.

The test design property receives the maximum rating of 5 as the knowledge of the project resides with the tester which is also the sole developer of the project.

The test environment property receives a rating of 3 as compromises have been made to the similarity of the testing environment to the production environment. These compromises have been made to reduce costs but in return reduce the representativeness of the testing. Ideally one would want a carbon copy of production environment for use in testing to have a maximum amount of representativeness.

Both the feedback and autonomy properties get a maximum rating of 5 as the testing environment is fully automated and the test cycle time is less than an hour.

The reporting property gets a rating of 3 as there is a lack of tooling for code coverage. Ideally one would want a code coverage as high as possible as it is a measure for the rigour of testing.

Project of SIG The final rating of the project of SIG is 3.5, which is the aggregation of the four testing quality characteristic ratings as shown in table 4.1.

The testing quality characteristics ratings are an aggregation of the test quality property rating as summarised in figure 4.2. We explain the scores for each of the testing properties.

The project scored a rating of 3 on the test scope property, this is due to a lack of prioritisation of the testing effort. As with the project from Schuberg Philis, everything was tested with equal importance as it was part of the definition of done for all features. According to the testing model, a good practice would be to prioritise the testing based on how much effort. This prioritisation is needed as it is impossible to fully test everything in a project.

The project was rated a single star due to the lack of recording the requirements. In the software defined infrastructure project of SIG, the requirements are implicitly known by the developers and testers but are not made explicit. According to the testing quality model, gathering requirements is a good practice as it allows linking requirements to functional tests.

The test design property was rated with the maximum rating of 5 stars as the infrastructure tested is also the infrastructure the testers use in their day-to-day activities.

The test environment property is rated with 3 stars as the testing environment is not an exact copy of the production environment. The team has decided that the testing environment consists of only one server of teach type of role they have defined. It is considered a good practice to have the testing environment imitate the production environment.

Both the feedback and autonomy properties get a maximum rating of 5 as the testing environment is fully automated and the test cycle time is less than an hour.

The reporting property is rated with 2 stars due to a lack of requirement coverage. As requirements are not gathered requirement coverage can not be measured. It is considered a good practice to test each requirement as a test of functional completeness.

Limitations found by application

We feel that the ratings of the Schuberg Philis and SIG projects are too near to each other. The difference in total rating is just 0.4. We think that this difference is too low because the project from Schuberg received a too low score. We feel that is legitimate when applying the DevOps philosophy that there is a definition of done stating that every feature should be tested before going into production and that no prioritisation by testing effort has to be made. Change requests are not really a thing when applying an agile methodology as changes are encouraged by having a product owner close to the development team, inserting changes with every iteration of the development cycle. A fix of these issues should increase the rating of Schuberg while the rating of SIG should stay roughly the same as the major difference between the two projects is the lack of explicit requirements within the project of SIG.

We feel that the testing model is catered a bit towards the waterfall

development methodology and not really towards agile. This is evident in the wording of certain questions, such as asking for formalised requirements or formalising the change requests.

At one point we found that the sequence of questions is off, as a question about the time it takes to find the root cause of a problem is dependent of a question about coverage in the model. Both projects have answered negatively to the coverage questions while they would have had a positive outcome on the time it takes the root cause.

The thresholds on questions relating to autonomy and feedback time are too narrow. In a software defined infrastructure context, where testing is quite often automated, a test cycle time of two weeks is unheard in of our opinion. As automation of testing is spread out in this sort of environment, scoring high on these properties would be too easy.

4.2.3 Relevance of themes of the test quality model

Another approach for finding limitations of the current test quality model is by interviewing experts on software defined infrastructures. We wanted to know which themes covered by the current test quality model were relevant in the context of software defined infrastructures. To find out we provided the experts with a list of themes covered by the current model and asked them to rate each theme based on its relevance. The scale of rating of the themes was based on a one-to-five Likert scale where one stands for a very low degree of relevance and five stands for a very high degree of relevance. The questionnaire can be found in appendix D.

Figure 4.3 displays the result of the themes questionnaire. The chart is sorted based on score, from a high relevance score to a low relevance score.

These results show that the experts feel that replicating the production environment in a test environment is very important and that tests should be automated where possible. This is likely due to the nature of software defined infrastructures projects being infrastructure automation projects. When the production environment is automated, it should not be too hard to replicate that environment and run tests against it. Automation is very important due to the agile nature of these projects where changes in code are tested often in short iterations.

The results also show that the experts feel that the planning and scoping of testing is not very important. Their opinion is that everything in a software defined infrastructure project should be tested and that therefore no planning is required. Another thing the experts did not find important was the handling of change requests and traceability of tests to requirements. All the current requirements should be tested anyway and a formal change request policy adds redundancy, as in an agile type project requirements tend to change a lot anyway. Traceability of tests to requirements is not important as there should be more tests than just those for the requirements to cover

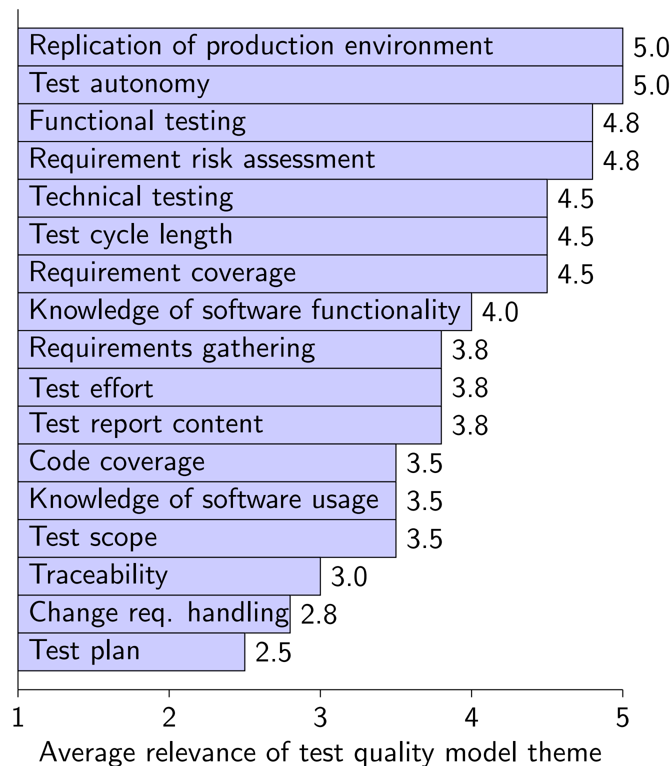

Figure 4.3: Result of questionnaire measuring the relevance of the themes

the unhappy paths, which are usually not documented in the requirements.

4.2.4 Acceptance of the test quality model

As a last step, a questionnaire to measure the acceptance of the currently quality model was given to the experts.. With the questionnaire we wanted to find out if the method of conduction of the current test quality model is accepted by the experts. The questionnaire is loosely based on [20] in which acceptance of methodologies is among developers is tested. Below is a list of questions together with the acceptance criteria the questions are measuring. The answer possibilities are based on a Likert scale. We provided five check boxes ranging from a very low degree to a very high degree. All the questions had a text box where the experts could provide input on why they gave the answer that they did. The questionnaire can be found in appendix D.

Usefulness To what degree do you think this model is useful when measuring the testing quality of an SDI project?

Easy of use To what degree do you think using this model would require much effort?

Compatibility To what degree do you think this model is consistent with testing practices in SDI environments?

Volunatariness If you were to assess the testing quality of an SDI, to what degree do you think you would use this model in the future?

Measurability To what degree do you think this model is able to measure the testing quality of an SDI project?

The last question asked in the interview was an open questions asking if the expert feels that there are aspects missing about testing in an SDI project in this model.

Figure 4.4 shows the result of the questionnaire. The chart shows the average result of each question and is sorted from high score to low. It should be noted that the score for ease of use is inverted as the question was asked in a negative manner.

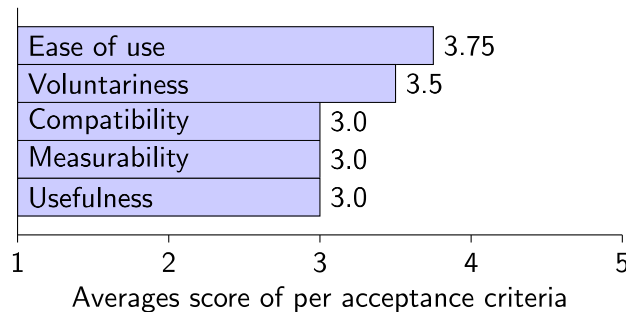

Figure 4.4: Result of questionnaire measuring the acceptance criteria

These results indicate that the experts think that the model itself is easy to use and that they would use a model that is shaped like the current test quality model in a software defined infrastructure context. The slightly lower score on compatibility, measurability and usefulness indicate that the experts are not convinced that the test quality model in its current shape is fit to be used as model for measuring the testing quality in a software defined infrastructure.

4.2.5 Issues of the current model

In this section we summarise the issues found in the current test model. These issues are either a direct result from the application of the model or the interviews. Another source of issues is the insight we gained while doing the application and interviews, and during the preparation and rounding up of both. We split the issues up in two categories. One category where we summarise issues we found in the model that are not specific to software defined infrastructures. And another category where we summarise issues

related to the application of the model in a software defined infrastructure context. The separation of these two issues help at identifying general improvement and improvement for the model in relation to software defined infrastructures.

General issues with the model

Mapping of properties with characteristics In the current model there is a mapping between the report property and the representativeness of testing quality characteristic. During a discussion with one of the experts we came to the conclusion that the mapping does not make sense as the report of a test has no relation with the representativeness of that test. This wrong mapping gave us the impression that other mappings, or lack of mappings might be wrong as well.

Wording of concepts We feel that some concepts are incorrectly worded or are addressed with vague terms. An example of that is the testing property 'test design' which includes does not includes any questions relating to the design of the tests but rather relating to things that might influence the test design.

Mapping of questions inside properties To measure the reporting property, the model asks questions about coverage of tests. In a discussion with one of the experts we found that this was odd. We understand that coverage might be measured and reported, but is to us an issues related to test design rather than the test report.

Catered towards waterfall methodology The current test quality model speaks of a formalised requirements and change request processes. We feel that in projects using a more agile approach, where requirements gathering is a less formal process, tests can still be based on the functionality that is described explicit but less formal requirements.

Traceability from test to requirement In a discussion with one of the experts it was brought up that you should have more tests than solely specified in the requirements. Such is the case for unhappy-path testing where you test edge cases typically not described by the requirements. Even though the test suite is effective in this case, the score would be lower as not every test would be linked to a requirement.

Unhappy-path testing Some of the experts would like to have a sense of coverage of unhappy-path testing to know when they have tested enough

cases. The model describes unhappy-happy path testing rigour as an implication of the technical and function knowledge of the software of the testers, yet does not ask specifically whether there are enough unhappy-path tests. We feel that there should be a question or a set of questions specifically addressing unhappy-path testing as we think it unhappy-path testing is an absolute necessity. This is certainly the case in a software defined infrastructure project, where you would have to test for potential security issues, such as for unintended users being enabled in the operating system.

Hard to achieve thresholds At one point the model asks if the requirements coverage is 100%. One of the experts pointed out that this is too steep and that the value should be lowered.

Issues with the model in relation to SDI

Wide threshold ranges In line with the model being catered towards the waterfall methodology, we feel that the ranges for the test cycle time are set too wide. A project can still earn more than a single star in this property if the test cycle takes less than two weeks. We feel that two weeks is way too much and that therefore it is too easy to get a good score on this criteria. The same issues goes for the level of automation which is measured in man-hours. A project can earn more than a single point here if there are less than ten man-hours involved in running a single test cycle. We feel that with the possibility of fully automating the test suite, less than ten man-hours is too easy to reach.

Testing of different languages One of the experts pointed out that software defined infrastructure projects usually consist of more than one technology. The code to provision the servers is usually written in another language than the code to launch and shut down virtual machines. The expert said that developers should be wary to not just test code of the main language but also the smaller pieces written in other languages.

Convergence of configuration Configuration management tools try to bring a server into a certain state. One of the issues the SDI team of SIG ran into was that servers sometimes were not able to reach the desired configuration when the they were installed from the ground up, but could reach the desired state if they already were in a previously desired state. In order to test if an infrastructure is able to configure itself after a complete restart, one would have to also test if the servers are able to configure themselves from the ground up.

Lack of emphasis on automation All the experts have pointed out that automation of testing is really important as it allows to test more efficiently,

allowing for more development iterations. Other than efficiency automated tests are also more effective as they are less prone to human-error.

Co-existence of test and production Another issues that the SDI-team of SIG ran into was that certain applications would probe the network themselves to find out which server or client to connect to. In a concrete example, this has lead to the server responsible for creating back-ups of data in the production environment, to contact machines that were residing in the testing environment. Ideally, the test environment should be designed so that this can not happen. Other than the design of the test environment, one could also write tests to detect cross-environment connection possibilities.

4.3 New testing quality model

In this section we describe the testing quality model made for software defined infrastructures[21]. The testing quality model for software defined infrastructures is based on the confidential SIG testing quality model[22].

4.3.1 Testing quality characteristics

We see no reason to adjust the quality characteristics of testing. These quality characteristics are based on the ISTQB foundation level syllabus and we feel that diverging from the syllabus means would diverge from an industry standard. One minor change that we would have wanted to do is renaming the evolvability characteristics into 'adaptability' Evolvability can be defined as: "changes in a systemś environment (domain), requirements (experience) and implementation technologies (process)" [10]. We feel though that the term evolvability implies natural change and not changes made by developers. We would have preferred the term adaptability, which is defined by the Oxford Dictionary as: "Able to be modified for a new use or purpose"[4] which fits the context of changes made to tests by developers better. Because evolvability is already a term used by the industry, we refrain from implementing the word adaptability.

4.3.2 Testing quality properties

Renamed test design

Renamed the test design property to 'Software knowledge'. The description of test design was: "The extent to which the tests resemble the real world situation in production is estimated and for the evaluation of this property". What this meant in practice is that the decision tree inside the property was asking questions related to the knowledge of the testers about the software

[4]http://www.oxforddictionaries.com/definition/english/adaptable

so that they can use that knowledge as input to design effective tests. The questions therefore are representing if some of the pre-conditions of a good test design are being met but not if the test design itself is of a high quality. We do agree with the SIG testing quality model that a weak knowledge about the software impedes the representativeness, so the property should remain. The description of the software knowledge has been changed to: "The extent to which the testers have enough knowledge about the software product, allowing them to design demanding tests". With these changes we aim to improve the overall correctness of the definitions of the testing quality model, which increases the accuracy of the mappings of the mappings from testing quality properties to testing quality characteristics.

New property: test design

As we renamed test design to software knowledge, we felt that there is now a category missing measuring the quality of the design of the tests. This property should measure to which extent the test suite covers artefacts such as requirements and code and if there is enough rigour in testing unhappy paths. The description of the property is: "The extent to which the test suite covers both the artefacts of the software product, as well as the desired and undesired functionality of the software system". With this new property resolve the issue that there is no notion of unhappy path testing in the SIG testing quality model. Coverage of requirements and code were already present in the model but were placed inside other properties causing those properties to have odd property to characteristics mappings. The details of the mappings is explained in subsection 4.3.3. The details of the decision tree is explained in subsection 4.3.5.

4.3.3 Mappings of properties to characteristics

Because of all the changes we have made and because we disagreed with some of the original mappings we decided to remap the testing quality properties to the testing quality characteristics. A summary of the mappings is depicted in figure 4.5.

Mapping per property

We provide, for each of the testing quality properties that we have now defined, the argumentation for the mappings to the testing quality characteristics. Good mappings of the testing quality properties to testing quality characteristics improves the accuracy of the result as the testing quality characteristics are being represented better.

Test plan Test plan influences all four quality characteristics as it is the fundament of the testing process. The representativeness is affected by

	Test Plan	Test Validation	SW Knowledge	Test Environment	Feedback Duration	Test Automation	Test Reporting	Test Design
Representativeness	X	X	X	X				X
Effectiveness	X		X	X		X	X	X
Efficiency	X				X	X		
Evolvability	X	X	X	X	X			

Figure 4.5: New Mappings of test quality properties to test quality characteristics

the scoping of the testing. If the scope is smaller than the whole of the project, then representativeness is impeded. Effectiveness and efficiency are affected by the determination of the testing approach. Choices made in the testing approach could include tooling and the rigour of tests, including coverages. Adaptability is affected by determination of how to deal with changing requirements.

Test validation Test validation influences the representatives and adaptability characteristics. Representatives is affected by having the requirements explicit. Explicit requirements can act as input for the creation of functional requirements. Adaptability is affected as requirements are bound to change, meaning that functional tests should be adapted as well.

Software knowledge The software knowledge property influences the representativeness, effectiveness and adaptability characteristics. A good knowledge of the software system implies that testers able to write tests that represent the use of the system. Knowledge also affects that testers are able to determine edge cases better causing the tests to be more effective at finding bugs. Adaptability is affected as testers are likely to understand changes in the functional requirements better, allowing them to change the existing tests more effectively.

Test environment The software knowledge property influences the representativeness, effectiveness and adaptability characteristics. Replication of the production environment to the test environment makes the testing process more representative. The effectiveness of the tests is increased as the testing suite will be more likely to find errors caused by the software product and not by the confounding factors of the differences in the test environment. The testing environment should allow being adapted when requirements or changes in the production environment happen.

Test feedback The test feedback time property affects the efficiency and evolvability characteristics. Shorter test feedback time increases the efficiency as less time resources are consumed. Evolvability is affected as short feedback time increases the feedback testers receive allowing them to change tests when earlier tests were found to be erroneous.

Test automation The test automation property influences the effectiveness and efficiency characteristics of testing. Effectiveness of the tests is increased as a higher level of automation leaves less room for human error. Efficiency is affected as automation of tests frees up developers' time.

Reporting The reporting property influences the effectiveness quality characteristic of testing. Effectiveness is affected a good quality of test reporting increases the ability of developers to find the root cause of a failing test.

Test design The test design property influences the representativeness and effectiveness characteristics of testing. A good test design covers most, if not all of the functional requirements, making the testing suite representative. A good test design also covers most, if not all of the code, making the tests more able to detect faults.

4.3.4 Aggregation of ratings

We changed the aggregation of testing properties. The new system does not give out stars for the testing properties, but still gives out stars for the testing quality characteristics. Reason for the is that we felt that giving out a star rating for the properties creates a tendency where decision trees are made in such a way that they can provide a one, two, three, four or five star rating. This makes it hard to make decision tree that do not have five questions or a multiple of five questions.

Each decision tree can now get a score from in the range of [0,1]. Zero means that all decision tree ended right away due to negative answers in the first question. One means that all decision tree have been successfully traversed, but having all questions positively answered. Each decision gives a fraction, such as $1/3$. The fractions given by each decision tree are then added up to give a final score for that property.

Then, like in the already existing testing quality model, the rating for each property are aggregated to the quality characteristic by averaging them. The rating of a property has a minimum of one and a maximum of five. To achieve this, the following formula is used:

$$\text{Rating}_{\text{characteristic}} = 1 + 4(\text{rating}_{\text{properties}})$$

The final testing quality rating is then the average of the ratings of all the characteristics. To translate the score into a SIG star rating, the rating is either rounded towards the nearest round number or to the nearest half representing the number of stars to be given.

4.3.5 Decision trees inside the properties

In this section we explain the changes we made to the to the decision trees of each testing quality property. The decision trees can be found in the new testing quality evaluation model [21].

Test plan The test plan property was named test scope in the original testing quality model. The goal of this property is to find out if there is a test plan that contains the test approach, scope and prioritisation of testing. A change in the decision tree is the removal of the risk assessment for requirements questions. We feel that the risk assessment of the requirements is part of the validation of test validation property and how that risk assessment is then being used should be part of the test plan.

Test validation The goal of the test validation property is to assess if requirements are explicitly known. Wording of the questions has been changes so that the model does not require formalised requirements, which does not fit the agile development methodology. Formalising of change requests has been removed to generalise methodologies as well, in agile environments there usually is no change request process as changes to requirements can be made at any time without starting a process. Requirements traceability has been removed as we feel that not every test has to have a requirement. There can be unhappy-path tests testing edge cases for which no requirement has been made, but are still included by an experienced tester to discover common mistakes.

Software knowledge The software knowledge property is what used to be the test design property in the original testing quality model. The only change made to this tree is that the question relating to functional testing has been removed as that was the only question having to do with test design. The rest of tree remains the same except for some changes of wording.

Test environment The test environment tree has had two changes. The splits up the questions about the level of replication of the test environment and adaptability of the test environment. We feel that these questions are independent of each other so they are now two different paths within the decision tree. Determining the quality of replication of the environment has been worded more loosely. In the original test quality model the replication

of the test environment was had specific wording such as replication of the topology. We generalised this allowing the person conducting the model to decide the level of replication. This allows us to account for differences in the test environment that are deliberate.

Feedback time We only changed the thresholds for the feedback time property. These changes are made as the test cycle time and the feedback delivery thresholds were too easy. In a time where testing can be almost fully automated we feel that we can make these thresholds more strict.

Test automation As with the feedback time property, we reduced to thresholds of the test automation property. We feel that in a time where tests can be fully automated we can make the threshold more strict.

Reporting The test reporting property contained questions referring to code coverage and requirement coverage. We feel that these questions are related to test design, as they are measures of how effectiveness of the test suite. Other than removal of the coverage questions and some changes in the wording, the reporting property has not changed.

Test design Test design is a new property. It tries to determine the effectiveness and the representativeness of the tests. Measures for code coverage and for the amount of unhappy-path testing are measures for the effectiveness of the tests, as they determine the amount of the software artefacts that are being covered by the tests. Requirement coverage measures the representativeness of the tests, as in the ideal case, every requirement should be tested before a piece of software is shipped.

4.3.6 Application of the new model

	Project of	
Characteristic	**Schuberg**	**SIG**
Representativeness	3.8	2.7
Effectiveness	4.2	3.8
Efficiency	4.6	4.6
Evolvability	4.0	3.5
Final rating	4.1	3.6

Table 4.2: Result of application of the new testing quality model on two projects

We performed a test run based on the data gathered in the application of the original testing quality model as described in section 4.2.2. The results of applying the data to the new test model can be found in table 4.2.

As the data we gathered for application of the original testing quality model is not sufficient to answer all the questions in the new model, we made assumptions for the questions for which answers were missing. We assumed that the project from Schuberg has sufficient amount of unhappy path and we assumed that the project of SIG has an insufficient amount of unhappy path testing. When interviewing the experts on the relevance of the original testing quality model, described in section 4.2.3, one of the SIG interviewees expressed that he is not yet happy with the amount of unhappy path testing. The interviewee from Shuberg Philis expressed being happy with his whole testing approach.

Difference with original TQM

	Project of	
Characteristic	**Schuberg**	**SIG**
Representativeness	+0.6	-0.1
Effectiveness	+0.5	+0.5
Efficiency	-0.4	-0.4
Evolvability	+0.3	+0.5
Final rating	+0.2	+0.1

Table 4.3: Differences of the old TQM compared to the new TQM

Table 4.3 shows the differences in the results between the application of the original testing quality model versus the new testing quality model. For each cell the difference is calculated by subtracting the result of the original model from the result of the new model.

The table shows that the ratings of each of the projects has increased and that the difference between the project has also increased. This is in line with our thought that the ratings were too low and that the difference between the project was too narrow.

The score for representativeness has increased for the Schuberg project while it has decreased for the SIG project. This result can be accounted to the new test design property in which the Schuberg project scores higher than the average representativeness of the original model, while the SIG project scores lower than the average representativeness of the original model.

The effectiveness rating increases for both project in the new model. This increase in score can be accounted to the new mappings as the test automation and test environment properties are now mapped to the effectiveness characteristics. Both projects get a high rating for these properties,

which then propagated to the effectiveness characteristic.

The both projects get a lower rating on efficiency which can be accounted to the new mapping of the test design property to the efficiency characteristic. Given that they had a perfect rating in the original model and that both projects received a sub-perfect rating for test design, thy both show a minor decrease.

Both projects show an increase in evolvability when applied to the new model. The increase for both projects is because of the inclusion of more quality properties in the evolvability characteristic. Both projects have a high rating for feedback duration which now influences evolvability. The increase is steeper for the SIG project due as the relative weight of the test validation property to the evolvability characteristic in which the SIG project was rated lower than the Schuberg project.

The results are a step in the right direction, though we think that the difference of the final ratings for the project should be a little further apart. We believe that the lack of requirements gathering of the SIG project is harmful for the functional testing of the infrastructure, which we believe is important. An improvement to the new model could be adding weights to either the properties or the characteristics.

4.4 Conclusion

During the interviews for gathering important quality aspects for software defined infrastructures as described in chapter 3, we found out that testing is an important quality aspect of software defined infrastructures.

In this chapter we created a model for measuring the quality of testing of software defined infrastructures. To come to a model for measuring software defined infrastructures, we applied an already existing testing quality model created by the Software Improvement Group to projects. We also interviewed experts on software defined, about their acceptance of the existing testing quality model, and about which themes they feel are relevant and irrelevant to testing in software defined infrastructures. We used the results of the application of the existing model and the interviews as input for changing the existing model into one that is more compatible with software defined infrastructure practices.

Chapter 5

Conclusion

In this chapter we conclude the thesis. We provide an overview of the results and contributions of the thesis and we end by providing pointers for future work opportunities in the field of quality evaluation of software defined infrastructures.

5.1 Results

This section gives an overview of the answers to the research questions and of the contributions of this thesis.

5.1.1 Answers to research questions

RQ1: How can we define quality of software defined infrastructures?

The first goal of this thesis was to create a quality model for software defined infrastructures. To create a quality model we had to figure how to identify quality aspects and how to create a model from the identified quality aspects.

We reviewed literature on the creation of quality models in the field of software engineering. Prior studies on software quality identified quality aspects either trough the study of literature or by consensus of practitioners. We applied a method for identification similar to the consensus approach. We interviewed practitioners of software defined infrastructures from Schuberg Philis and the Software Improvement Group to assess what these practitioners think is important when assessing the quality of software defined infrastructures. The outcome of these interviews can be found in section 3.2.2. The outcome of the interviews suggests that among other important quality characteristics, testing is a very important quality characteristic.

The outcome of the interview was used to construct a quality model for software defined infrastructures. This quality model can be found in section

3.3.2. The model decomposes quality of software defined infrastructures into software quality, deployment process quality and infrastructure quality. For each of these decompositions quality characteristics have been defined.

RQ2: How can we evaluate the testing quality of software defined infrastructures?

The second goal of this thesis was to create a model for the evaluation of testing quality for software defined infrastructures. Testing was found to be an important quality aspect of software defined infrastructures in the first part of this thesis.

The Software Improvement Group has a quality model for the evaluation of testing for traditional software development projects [22]. We applied the testing quality model to two software defined infrastructures projects. The first project was from Schuberg Philis, a company that creates computer infrastructures for external clients. The second project was the internal software defined infrastructure from the Software Improvement Group and is used to deliver applications to its lab.

The results from applying the SIG testing quality evaluation model can be found in section 4.2.2. The results were an indication that the model had a limited ability to measure the quality of testing of software defined infrastructures.

To create a model that was more suited for measuring the quality of testing of software defined infrastructures, we decided to ask practitioners what themes of the SIG testing quality model they thought were relevant, and which they thought were irrelevant. The outcome of the interviews can be found in section 4.2.3. The results indicate that automated testing and replication of the testing environment are the most relevant themes, while planning of tests and the handling of change requests were considered the least relevant themes.

Next to identification of relevant themes we also looked at the voluntariness of adoption of the SIG testing quality model. We asked the practitioners questions related to the ease of use of the SIG testing quality model and its ability to measure testing quality of software defined infrastructures. The outcome of this interview can be found in section 4.2.4. The outcome suggests that the practitioners think that the SIG testing quality model is easy to use and that they would use it for software defined infrastructures if it was compatible with the current practices in software defined infrastructure development.

The outcome of the interviews on the testing quality evaluation model provided was then used as input for the creation of a model more compatible with SDI practices. The new testing quality evaluation model [21] is less strict on metrics evaluating requirements, but is more strict on processes which can be automated. This is in line with DevOps practices which are

applied in software defined infrastructure projects. An overview of changes made can be found in section 4.3.

The new testing quality evaluation model was validated by reapplying the data gathered for the application of the original testing quality evaluation model. The results of the validation can be found in section 4.3.6. The validation indicates that the new testing quality evaluation model is more compatible with existing software defined infrastructure practices.

5.1.2 Contributions

The goals of this study was to create a quality model for software defined infrastructures and to create a quality model for the evaluation of testing quality of software defined infrastructures. The contributions of this thesis are:

- The identification of quality aspects of software defined infrastructures as seen by practitioners.
- The definition of a quality model for software defined infrastructures. This quality model can be used as framework for the creation of metrics or evaluations models to measure the quality of software, the quality of the deployment process and the quality of the resulting infrastructure. The combination of these parts represent the quality of the overall software defined infrastructure.
- The analysis of the SIG testing quality evaluation model. This analysis consists of determination of the relevance of the themes covered by the model in relation to software defined infrastructures and consists of the determination of the degree of adoption of the model on the context of software defined infrastructures.
- The definition of a testing quality evaluation model for software defined infrastructures. This model can be used to determine the quality of the testing process of software defined infrastructure projects.
- The validation of the testing quality evaluation model for software defined infrastructures by reapplication of project data gathered in earlier stages of the study to the newly created model. The validation shows that when applying the new testing quality evaluation model, results are closer to what we think is the actual testing quality of the projects.

We hope that these contributions will act as a foundation for the research on quality assessment of software defined infrastructures. We feel that they can help in providing an objective judgement on what quality in software defined infrastructures are and we hope that it will contribute to maturing the overall field of software defined infrastructure creation.

5.2 Future work

We provide some pointers for possible future work extending this study and to other topics in the field of software defined infrastructures.

5.2.1 Extending this study

We identified several ways to extend the study done in this thesis:

Enlarging the data sets

The gathering of data for this thesis was limited to practitioners and projects of only two companies, namely The Software Improvement Group and Schuberg Philis. This limitation could mean that the data gathered was influenced by the cultures of both companies and might impact the generalisability of the results. Enlarging the data set, by stretching it over more companies and more practitioners, could reduce the impact of company culture in the results.

Refinement of model creation

The process of creating a quality model includes the identification of quality aspects for the process or artefact for which the quality model is created. When quality aspects are identified they need to be transformed into a quality model by a process of eliminating and grouping quality aspects. Prior research shows that these identification and transformation steps are reliant on the discretion of the creator of the model or are based on consensus. This causes the identification and transformation steps to be influenced by the perceptions of the creators which could impact the accuracy and generalisability of the resulting model. Future research that makes use of more objective and quantifiable approaches could eliminate the influences of the perception of the creators of the model and thus increase the generalisability.

Validation of the models

The testing quality evaluation model resulting from this study has been validated using data gathered for the application of the SIG testing quality model. Application of the new model using existing data required assumptions to be made which results in some uncertainty in the validation of the newly created models. Research validating the model could include probing practitioners for their opinion on the model and applying the models on more project to see if the ratings yielded by the model are approaching expected ratings.

Calibration of the models

The models resulting from this study assume that the parts of the model carry an equal weight in the quality of artefact or process that is to be evaluated. It is currently unknown which parts of the models carry more weight in the overall quality of software defined infrastructures and testing of software defined infrastructures. Calibration of the model, for example by applying the model to more projects, could help in either creating a normative approach for rating, meaning that the resulting rating is relative to the ratings of other projects.

5.2.2 Research on software defined infrastructures

As discussed in section 3.1.1, the field of software defined infrastructures is relatively new and has yet to be picked up by the academic community as a field of research. We identified the following interesting subjects related to software defined infrastructures that could be explored:

Effectiveness of software defined infrastructures

While software defined infrastructures are being used, there is, to our knowledge, only anecdotal evidence that application of software defined infrastructures leads to more effective IT infrastructure management. We see opportunities in comparing software defined infrastructure to other methods of infrastructure management. An example would be to investigate what size an infrastructure should be before automation using software defined infrastructures becomes more effective than traditional approaches.

Applicability of software engineering practices

Because the body of scientific knowledge on software defined infrastructures is small, one could look at fields that have an overlap with the field of software defined infrastructures to look for best practices. An example of such a field is the field of software engineering. Because of this overlap one could look at the extensive knowledge that exists on software engineering and test the applicability of that knowledge to software defined infrastructures.

Bibliography

[1] Rafa E Al-Qutaish. "Quality models in software engineering literature: an analytical and comparative study". In: *Journal of American Science* 6.3 (2010), pp. 166–175.

[2] Michael Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[3] D. Athanasiou et al. "Test Code Quality and Its Relation to Issue Handling Performance". In: *Software Engineering, IEEE Transactions on* PP.99 (2014), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2342227.

[4] Antonia Bertolino. "Software Testing Research: Achievements, Challenges, Dreams". In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–103. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.25. URL: http://dx.doi.org/10.1109/FOSE.2007.25.

[5] Barry W Boehm, John R Brown and Myron Lipow. "Quantitative evaluation of software quality". In: *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press. 1976, pp. 592–605.

[6] Lionel Briand. "Embracing the engineering side of software engineering". In: *Software, IEEE* 29.4 (2012), pp. 96–96.

[7] Mark Burgess. "Configurable immunity for evolving human–computer systems". In: *Science of Computer Programming* 51.3 (2004), pp. 197–213.

[8] Mark Burgess. "On the theory of system administration". In: *Science of Computer Programming* 49.1 (2003), pp. 1–46.

[9] Ben Cherian. *What Is the Software Defined Data Center and Why Is It Important?* (Retrieved on 23-04-2014). 2013. URL: http://allthingsd.com/20130613/what-is-the-software-defined-data-center-and-why-is-it-important/.

[10] Selim Ciraci and Pim Broek. "Evolvability as a quality attribute of software architectures". In: (2006).

[11] Edsger Wybe Dijkstra. *Notes on structured programming.* Technological University Eindhoven Netherlands, 1970.

[12] Open Network Foundation. *Software-Defined Networking: The New Norm for Networks.* (Retrieved on 23-04-2014). 2012. URL: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf`.

[13] Ilja Heitlager, Tobias Kuipers and Joost Visser. "A practical model for measuring maintainability". In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the.* IEEE. 2007, pp. 30–39.

[14] Waldemar Hummer et al. "Testing idempotence for infrastructure as code". In: *Middleware 2013.* Springer, 2013, pp. 368–388.

[15] Coraid Inc. *The Fundamentals of Software-Defined Storage: The Fundamentals of Software-Defined Storage.* (Retrieved on 23-04-2014). 2013. URL: `http://san.coraid.com/rs/coraid/images/SB-Coraid_SoftwareDefinedStorage.pdf`.

[16] *ISO 25010 - Systems and software engineer - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.* Tech. rep. International Organisation for Standardisation (ISO), 2011.

[17] Jim A McCall, Paul K Richards and Gene F Walters. *Factors in software quality. volume i. concepts and definitions of software quality.* Tech. rep. DTIC Document, 1977.

[18] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing.* Tech. rep. National Institute of Standards and Technology, 2011. URL: `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[19] Thomas Mller et al. *Certified Tester Foundation Level Syllabus.* Tech. rep. International Software Testing Qualifications Board, 2011. URL: `http://www.istqb.org/downloads/finish/16/15.html`.

[20] Cynthia K. Riemenschneider, Bill C. Hardgrave and Fred D. Davis. "Explaining software developer acceptance of methodologies: a comparison of five theoretical models". In: *Software Engineering, IEEE Transactions on* 28.12 (2002), pp. 1135–1145.

[21] *SIG Testing Quality Model, SDI Version 0.1.* Tech. rep. Software Improvement Group B.V., 2014.

[22] *SIG Testing Quality Model, Version 1.0.* Tech. rep. Software Improvement Group B.V., 2013.

Appendix A

Interviews SDI quality characteristics

A.1 Interview #1

A.1.1 Quality aspects of internal project

Testability is important. With testability I mean the existence of unit tests and integration tests. Integration testing means allowing the Puppet to provision a node and to let the integration testing suite run commands on the server to assert that the node is functioning. An example of such a test could be asserting that a node provisioned to be a web server that it actually serves a certain web page when requested.

There is no 1-to-1 mapping between Puppet code and integration test code. (Meaning that you integration tests do not fully cover the code).

The provisioning of a server is very fragile due to the many external dependencies, such the operating system of the machine that is to be provisioned. Software defined infrastructures is starting to have value when there is an automated test process validating the code. Testing is the most important aspect of software defined infrastructures as it is the only way you can check everything that is happening.

An issue we had with the internal project was that of a backup server that, while being set up in the testing environment was also backing up clients in the production environment. This resulted in the notion of coexistence. Which means in this case to take measures to ensure that nodes are not interfering with each other. This issue could be fixed by configuring the backup-application to not contact clients outside of its environment, or by enforcing that nodes are not able to reach each other beyond environment that they are in by creating rules in the network configuration.

The testing process of that project is by re-building the testing environment on a daily basis. Each morning at six o'clock the testing environment

gets built up from scratch to see if the code results in the desired infrastructure.

A.1.2 Quality aspects as an adviser

I have no experience with larger code bases.

One aspect that is important is that servers only provide is single service. Service in this case could be a something that consists of more than one application. For example a web application could be the combination of a web server and a database server. This in contrast to the hardware-centric approach of filling up a machine until all its resources are being used.

Usage of 3rd party libraries as-is is important in software defined infrastructure projects. This means not making changes to the library but either forking it as a separate project for maintaining it or by writing patches to give back to the open source project. The team itself has been sloppy on this in the past, adopting a library and therefore sitting with a large chunk of code in their own code base which they do not use themselves. These pieces of unused code provide support for operating systems they do not use within the company. For this they are currently discussing if they are going to remove all the unused code or if they are going to provide patches to the open source project of that library.

The libraries found on the internet (on Puppet Forge) generally do not carry very restrictive licences. Albeit that there is not a lot of jurisprudence on open source licenses, the company is not reluctant on using these libraries.

The development process of software defined infrastructures is more primitive than that of regular software development. This is due to the available tooling is less mature than compared to more traditional programming languages such as Java. The choice of tooling is important. This is due to the level of hobbyism involved in the community which causes many tools to be immature.

There are too many low level things programming have to keep in mind while developing software defined infrastructures such as creating configuration templates for different applications. The prediction is that in the future developers of applications will also provide deployment libraries so that deployment of those applications will be less operating system dependant.

The usage of an automated test and building process is important. This is in line with the continuous delivery approach, where building is done automatically on a daily basis or per-commit basis. With this the effect of the code becomes visible.

A.2 Interview #2

A.2.1 Quality aspects of internal project

Just like with other software projects it is important to follow the best practices.

Tooling is an important aspect, as software defined infrastructures are an area without a lot of history. The tooling is guiding you less, therefore the importance of best practices is larger.

Testing is important. Execution of of code is deployment, meaning that you move from unit testing to integration testing fast. The guarantee of unit testing is less than it is in regular software development projects, so you can only properly test by actually building up an infrastructure and running tests against it. Testing modules in isolation is not that useful.

Version control of 3rd party libraries is important. It is important to think about how you are going to use version control and how you are organising your code. Within software defined infrastructures, the usage of 3rd-party libraries is larger than it is in regular software development projects. This creates more dependencies on different channels. Within Puppet itself you already have dependencies on Gems of Ruby, the package manager of the operating system and Maven of Java. The dependency channels should be made clear so that you can see of how many channels you are dependent and that you can think about how you are securing/guaranteeing these channels.

In the internal project they are working on controlling the production environment. This is done with testing

As you are doing things remotely it is important to reduce the round-trip time as much as possible. This is done internally by not rebuilding the infrastructure all the time but just once a day in the morning. Next to that the developers have local sandboxes in which they can execute the code, but this only works well with nodes that do not communicate with other nodes.

A.2.2 Quality aspects as an adviser

As an external adviser, I would look for usage of supported tooling such as Puppet or Chef and see if they use a recent version of it.

For maintainability I would check if they have insight in their own code base and check if have automated testing. For static analysis important aspects are:

- Unit length
- Code complexity
- Volume

The technology stack should be small.

The usage of McCabe is troublesome in declarative languages and domain specific languages due the lack of if-statements. In these cases the usage of unit size as a complexity metrics is more useful.

Deployment automation is something to look at. The amount of manual operations being done on a server is something to look at.

Software defined infrastructures is where two worlds meet, Development and Operations (DevOps). Operations is tasked with providing reliability, guaranteeing a certain level of service. Developers on the other hand are occupied with reproducibility, automation and version control. As software defined infrastructures are the coming together of development and operations, it would be good to look at if a project is using the best of both worlds, that is:

- Are all artefacts under version control?
- Is there anything being done manually?
  - Is there continuous development / deployment?
  - Test automation?
- Tooling
- Monitoring
- Conservative with deployment of changes.

A.3 Interview #3

A.3.1 Quality aspects of internal project

It is important to look what at what the code looks like. Think of aspects such as:

- Files should not be too long
- Parameter lists should not be too long
- Identifiers got to have good names.

Long lists of parameters are fine in Puppet compared to typical object oriented languages. Though I do not yet have a good sense of when these things are too much.

The procedure of testing that I use is the following: I first tests thing on my laptop, then I test it in a sandbox, after that I put it in the staging environment.

Within unit test, you tests if resources are being there. But due to the declarative nature of the language, this is a bit senseless as you are testing very basal things. I think it is useful though to test the output of templates when the variables have been filled in. With unit tests you copy whatever you wrote down in the declarative language to the language of the testing suite. The integration tests are better. These bring a server up and start testing things such as the presence of a file.

Our infrastructure takes some time to get built. Because of that we build the infrastructure daily before work starts to run the integration tests on. An example for testing a service running a MySQL database would be to test if the MySQL process is running and to test if there is a response on the configured port.

Another important thing is following the standards made by the company creating the languages, which in the case of Puppet is PuppetLabs. The language subject to changes so it is important to keep track of the community and the announcements made by the company. The reason that the language is so subject to changes is due to the fact that only a single company is responsible, so it is very easy for them to make those changes if they deem it necessary.

A.3.2 Quality aspects as an adviser

If I would advise an external party I would formulate the previous better.

I would for example go look at how they separate code from data. Data in case of software defined infrastructures would configuration parameters such as which domain name servers (DNS) or network time protocol (NTP) servers should be contacted. In Puppet data can be separated from code using a tool called Hiera, which is a small database to store such data in.

Other things to look at would be whether or not there are tests and more specifically if there are integration tests.

Code smells would be something to look at, like imperative programming in a declarative language or more than necessary usage of if-statements.

A.4 Interview #4

A.4.1 Quality aspects of internal project

The Linux packages that we install and configure must contain the functionality that we need, so that we can stay as close as possible to existing installations. An example of this is the usage of the package manager and repository of Ubuntu in our case which contains tested packages, meaning that we do not have to test those packages any more. The management of the package is then being done by somebody else meaning that we do not have to provide specific versions of those packages as they are not bound to

change a lot. We only specify version numbers if there is a specific reason for us to install a newer version of an application than the one in the official repository.

One of the difficulties of software defined infrastructures is knowing how applications behave so that you know how to set them up using Puppet.

Another important aspect is the application of best practices such as not using exec statements too much as Puppet does not have control over the thing you do inside such a statement and cannot check if the result of the statement is correct.

Application states should be placed on a separate disk or volume so that it can be disconnected and be reconnected to another instance if needed. If possible this disk or volume should also be encrypted. The method for creating and mounting encrypted volumes is standardised and is called at many places inside to code as if it were an 'infrastructural design pattern'. In this case it is important to force a sequence of execution.

A.4.2 Quality aspects as an adviser

As an external adviser I would like to get a mental image of the infrastructure at-hand. How many servers are there, and are these instances of a single type of server or are these all different types? When having a homogeneous infrastructure you can easily switch off and replaces instances, while that is more difficult in a less homogeneous infrastructure. A less homogeneous infrastructure is also more prone to having unexpected thing happen.

Predictability could be a good aspect to look at. How did the client take care of the predictability? Predictability could be achieved for example by making a continuous deployment pipeline. Which is a development pipeline where code committed to a code repository is automatically unit tested, then it is deployed to a staging area where it is integration tested. After all tests succeed the code is placed in production. With Puppet you want this kind of safety when making changes.

You want to have an audit trail of all the changes, for example by making use of version control and by logging which versions have been deployed at which times and what the test results were of that deployment.

On the code level it is important to follow best practices. How close is the implementation to the how the tooling is supposed to be used? With Puppet this means to look at exec-resources and such.

A.5 Interview #5

A.5.1 Quality aspects of internal project

There are two dimensions of quality when it comes to software defined infrastructure projects. First there is internal quality, which is the quality of

your code, cookbooks or artefacts. Then there is external quality, which is the quality of the resulting infrastructure.

There is a lack of testing in software defined infrastructures as people tend to do main function testing, which is the sort of testing where people write a function and put it in the main function to see its outcome. Basically this sort of testing is testing the infrastructure like it is a C program. There are now people internally that do do unit tests on the cookbooks and integration tests on the cookbooks, cloud infrastructure and the cloud provider/orchestrator. There are selenium tests for front-end web testing. When you start generalising cookbooks for multiple operating systems, then unit testing becomes more complicated.

When creating cookbooks we keep extensibility in mind internally. We create our code in such a way that support for other operating systems can be added in an easy way.

We internally created a tool called 'Chefguard' to prevent developers altering versions of cookbooks without upping the version number. Not being able to make changes to versions is called 'version freeze'. It prevents certain side effects from happening due to having the same version number being used for different versions of the code. The typical scenario at this company is that when creating and testing code on a personal computer and then pushing it to a shared environment, such as the staging environment, the changed code does not cause unexpected behaviour on machines which are also reliant on cookbook containing that code. Another way of versioning the the infrastructure is by versioning the roles as well. Teams can decide for themselves if they wish to use newer versions of the role definition instead of the version being pushed automatically to every project.

At this company we pay close attention to not only keeping the testing environment and the production environment separated, but also keeping the infrastructures of the different clients separated, each having their own baseline creating a layered structured from company wide to team specific.

Code maintainability is an actual concern. We keep the code maintainable in an ad hoc manner. We do this by looking at the code over and over. I measure some of the maintainability metrics by using a Sonar Cube, but not everyone does that. They do revisit the code a lot due to us working in an Agile and Scrum-like manner, which keeps the code more maintainable.

Other projects, such as for example one that relies on Python keep maintainability high due to being forced to create tests. Having to make changes to your software and having to alter your tests over and over will help the developer realise how to write more testable code, which in turn is more maintainable.

A.5.2 Quality aspects as an adviser

As an external adviser I would start by looking at the level of maturity. Pointers for that would be things like creating automated builds using Jenkins, or having automatic infrastructure checks using Nagios, or by having using cloud orchestration tools such as Cloudstack or Openstack. Having these things would be an indication for a more mature infrastructure than when things are done manually.

From there I would help them trough the basics and the more advanced stuff. The basics are:

- Automation (which as extremely hard to do well)
- Being able to create stuff on the fly
- How do you test your automation?
- How do you monitor your systems?

At this company we have a tool to gather information and send out pages or SMS-messages to people on duty. This tools runs down a chain of responsibility until someone reports that they are picking up the problem.

After that I would look at the infrastructure that they are building. How is the infrastructure fitting with what you want to deploy out there? As you build and infrastructure to build an application on top of it.

It is a by-product of having many operations people doing these projects and not so many software engineers. Software engineers tend to make an effort of keeping things simple by making concious considerations of functionality versus effort. 'Is this feature worth the additional complexity?'

Here we create highly available infrastructures. To have highly available infrastructures means you have to have a redundant set of machines, either active-active, meaning that both of the sets share the load, or active-passive, meaning that one set of machines handles the everyday load and the other one sits idle acting as a fallback. This means you need to have load balancers and virtual IPs.

It is the same as with regular software, you want loose coupling and a high cohesion. Loose coupling in an infra for example by having a load balancer and by synchronising data.

When doing client projects this is harder because the infrastructure itself is a given.

You would have to think in levels of complexity and in levels of freedom. If you have all the freedom in the world then you should create a simple infrastructure for simple needs. If you do not have all this freedom then you have to create a complex infrastructure.

Client infrastructure are given meaning that it cannot be split up or have its dependencies swapped with comparable products, for example you cannot swap out an Oracle database with a MySQL database.

Appendix B

Reference topics and transcription

Audit trail	
Interview #4	You want to have an audit trail of all the changes, for example by making use of version control and by logging which versions have been deployed at which times and what the test results were of that deployment.

Automation	
Interview #2	The amount of manual operations being done on a server is something to look at.
Interview #4	...Which is a development pipeline where code committed to a code repository is automatically unit tested, then it is deployed to a staging area where it is integration tested. After all tests succeed the code is placed in production. With Puppet you want this kind of safety when making changes.
Interview #5	As an external adviser I would start by looking at the level of maturity. Pointers for that would be things like creating automated builds using Jenkins, or having automatic infrastructure checks using Nagios, or by having using cloud orchestration tools such as Cloudstack or Openstack.

Best practices

Interview #2	Just like with other software projects it is important to follow the best practices.
Interview #3	Another important thing is following the standards made by the company creating the languages, which in the case of Puppet is PuppetLabs. The language subject to changes so it is important to keep track of the community and the announcements made by the company.
Interview #4	Another important aspect is the application of best practices such as not using exec statements too much as Puppet does not have control over the thing you do inside such a statement and cannot check if the result of the statement is correct.

Co-existence

Interview #1	This resulted in the notion of co-existence. Which means in this case to take measures to ensure that nodes are not interfering with each other.

Complexity

Interview #5	You would have to think in levels of complexity and in levels of freedom. If you have all the freedom in the world then you should create a simple infrastructure for simple needs. If you do not have all this freedom then you have to create a complex infrastructure.

Data separation

Interview #3	I would for example go look at how they separate code from data. Data in case of software defined infrastructures would configuration parameters such as which domain name servers (DNS) or network time protocol (NTP) servers should be contacted.

DevOps

Interview #2	As software defined infrastructures are the coming together of development and operations, it would be good to look at if a project is using the best of both worlds,

Environments

Interview #1	This issue could be fixed by configuring the backup-application to not contact clients outside of its environment,
Interview #5	At this company we pay close attention to not only keeping the testing environment and the production environment separated, but also keeping the infrastructures of the different clients separated, each having their own baseline creating a layered structured from company wide to team specific.

Extensibility

Interview #5	When creating cookbooks we keep extensibility in mind internally. We create our code in such a way that support for other operating systems can be added in an easy way.

Functional suitabiltiy

Interview #5	After that I would look at the infrastructure that they are building. How is the infrastructure fitting with what you want to deploy out there? As you build and infrastructure to build an application on top of it.

Infra design

Interview #4	Application states should be placed on a separate disk or volume so that it can be disconnected and be reconnected to another instance if needed.
Interview #5	Here we create highly available infrastructures. To have highly available infrastructures means you have to have a redundant set of machines, either active-active, meaning that both of the sets share the load, or active-passive, meaning that one set of machines handles the everyday load and the other one sits idle acting as a fallback. This means you need to have load balancers and virtual IPs.

Level of freedom

Interview #5	You would have to think in levels of complexity and in levels of freedom. If you have all the freedom in the world then you should create a simple infrastructure for simple needs. If you do not have all this freedom then you have to create a complex infrastructure.

Maintainability

Interview #2 For static analysis important aspects are:

- Unit length
- Code complexity
- Volume

Interview #3 It is important to look what at what the code looks like. Think of aspects such as:

- Files should not be too long
- Parameter lists should not be too long
- Identifiers got to have good names.

Interview #5 Code maintainability is an actual concern. We keep the code maintainable in an ad hoc manner. We do this by looking at the code over and over. I measure some of the maintainability metrics by using a Sonar Cube, but not everyone does that. They do revisit the code a lot due to us working in an Agile and Scrum-like manner, which keeps the code more maintainable.

Maturity

Interview #5 As an external adviser I would start by looking at the level of maturity. Pointers for that would be things like creating automated builds using Jenkins, or having automatic infrastructure checks using Nagios, or by having using cloud orchestration tools such as Cloudstack or Openstack. Having these things would be an indication for a more mature infrastructure than when things are done manually.

Monitoring

Interview #2	As software defined infrastructures are the coming together of development and operations, it would be good to look at if a project is using the best of both worlds, that is: • ... • Monitoring • ...
Interview #5	From there I would help them trough the basics and the more advanced stuff. The basics are: • ... • How do you monitor your systems?

Node cohesion

Interview #1	One aspect that is important is that servers only provide is single service. Service in this case could be a something that consists of more than one application. For example a web application could be the combination of a web server and a database server. This in contrast to the hardware-centric approach of filling up a machine until all its resources are being used.

Predictability

Interview #4	Predictability could be a good aspect to look at. How did the client take care of the predictability? Predictability could be achieved for example by making a continuous deployment pipeline.

Test automation

Interview #1	The usage of an automated test and building process is important. This is in line with the continuous delivery approach, where building is done automatically on a daily basis or per-commit basis. With this the effect of the code becomes visible.

Interview #2	As software defined infrastructures are the coming together of development and operations, it would be good to look at if a project is using the best of both worlds, that is:
	• ...
	• Is there anything being done manually?
	  – ...
	  – Test automation?
	• ...

Testing

Interview #2	As you are doing things remotely it is important to reduce the round-trip time as much as possible. This is done internally by not rebuilding the infrastructure all the time but just once a day in the morning. Next to that the developers have local sandboxes in which they can execute the code, but this only works well with nodes that do not communicate with other nodes.

Testing	
Interview #1	Testability is important. With testability I mean the existence of unit tests and integration tests.
Interview #2	Testing is important. Execution of of code is deployment, meaning that you move from unit testing to integration testing fast. The guarantee of unit testing is less than it is in regular software development projects, so you can only properly test by actually building up an infrastructure and running tests against it. Testing modules in isolation is not that useful.
Interview #3	But due to the declarative nature of the language, this is a bit senseless as you are testing very basal things. I think it is useful though to test the output of templates when the variables have been filled in. With unit tests you copy whatever you wrote down in the declarative language to the language of the testing suite. The integration tests are better.
Interview #4	Which is a development pipeline where code committed to a code repository is automatically unit tested, then it is deployed to a staging area where it is integration tested. After all tests succeed the code is placed in production. With Puppet you want this kind of safety when making changes.
Interview #5	There is a lack of testing in software defined infrastructures as people tend to do main function testing, which is the sort of testing where people write a function and put it in the main function to see its outcome.

Testing effort	
Interview #4	An example of this is the usage of the package manager and repository of Ubuntu in our case which contains tested packages, meaning that we do not have to test those packages any more.

Tools usage	
Interview #1	The choice of tooling is important. This is due to the level of hobbyism involved in the community which causes many tools to be immature.
Interview #2	Tooling is an important aspect, as software defined infrastructures are an area without a lot of history. The tooling is guiding you less, therefore the importance of best practices is larger.

Version control	
Interview #2	As software defined infrastructures are the coming together of development and operations, it would be good to look at if a project is using the best of both worlds, that is: • Are all artefacts under version control? • ...
Interview #5	We internally created a tool called 'Chefguard' to prevent developers altering versions of cookbooks without upping the version number. Not being able to make changes to versions is called 'version freeze'. It prevents certain side effects from happening due to having the same version number being used for different versions of the code.

Appendix C

Application of TQM interview guide

Up front

- Let the interviewee summarise the project
- Try to find out how this project's testing approach differs from regular projects

Test scope

- Explain the testing approach of this project.
- Is there any testing done in this project?
- What is being tested?
- Is there a test plan? What does it look like?
- How important is testing in this project? How much testing is done in comparison to development?
- Is there a risk assessment for the requirements? How is such an assessment being done? When is it done? What is included in it?

Test validation

- Are requirements gathered and validated with the client? What do these requirements typically look like?
- What does the change request process look like?
- When the client makes a change request, what are the effects of it?
- How do you make sure that every test is sensible? How do you know there are not too many tests? Is every test traceable?

Test design

- What do technical tests look like in your project?
- What do functional tests look like in your project?
- Do you keep track of how the software is being used? How do you that?
- Do you keep track of technical aspects of the usage of the software in production? How?
- Do you test for unexpected behaviour? How do you do that?

Test environment

- What does the test environment look like? How does this differ from the production environment?
- What does the topology of the test environment look like?
- What is the difference in configuration between test and production environment?
- When changes to hardware or cloud service provider are being made, how does this affect the testing environment? What changes need to be made?

Feedback duration

- How long does a typical test cycle take?
- How is the result of the test delivered to the developers? How long does this take?

Autonomy

- How many man-hours are typically spent on a test cycle?

Reporting

- What does a test report look like?
- How do you make sure all the requirements are tested?
- Is code coverage a concern? How do you measure that? What was the last result?
- If a test failed, how would you look for the root cause? How long does that typically take?

Appendix D

Test quality model questionnaires

Themes questionnaire

Test topics questionnairre

July 30, 2014

To what degree is this a topic of importance in an SDI environment.

1 = Very low degree
2 = Low degree
3 = Average
4 = High degree
5 = Very high degree

1.	Change request handling	1	2	3	4	5
2.	Code coverage	1	2	3	4	5
3.	Functional testing	1	2	3	4	5
4.	Knowledge of software functionality	1	2	3	4	5
5.	Knowledge of software usage	1	2	3	4	5
6.	Replication of production environment	1	2	3	4	5
7.	Requirement coverage	1	2	3	4	5
8.	Requirement elicitation	1	2	3	4	5
9.	Requirement risk assessment	1	2	3	4	5
10.	Technical testing	1	2	3	4	5
11.	Test autonomy	1	2	3	4	5
12.	Test cycle length	1	2	3	4	5
13.	Test effort	1	2	3	4	5
14.	Test feedback time	1	2	3	4	5
15.	Test plan	1	2	3	4	5
16.	Test report content	1	2	3	4	5
17.	Test scope	1	2	3	4	5
18.	Traceability	1	2	3	4	5

Acceptance questionnaire

Feedback questionnaire TQM

July 30, 2014

Q1. To what degree do you think this model is useful when measuring the testing quality of an SDI project?	□ Very low degree □ Low degree □ Average □ High degree □ Very high degree	....
Q2. To what degree do you think using this model would require much effort?	□ Very low degree □ Low degree □ Average □ High degree □ Very high degree	....
Q3. To what degree do you think this model is consistent with testing practices in SDI environments?	□ Very low degree □ Low degree □ Average □ High degree □ Very high degree	....
Q4. If you were to assess the testing quality of an SDI, to what degree would you think you would use this model in the future?	□ Very low degree □ Low degree □ Average □ High degree □ Very high degree	....
Q5. To what degree do you think this model is able to measure the testing quality of an SDI project?	□ Very low degree □ Low degree □ Average □ High degree □ Very high degree	....
Q6. Do you think there are aspects about testing in an SDI project missing in this model	□ No □ Yes: Name the most important aspects missing.	....