Defining and measuring Puppet code quality

Eduard van der Bent

MSc Thesis (ICA-3583600)

June 21, 2016

Universiteit Utrecht

Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

External Supervisor:
Prof. dr. ir. Joost Visser
First supervisor:
Dr. Jurriaan Hage
Second supervisor:
Dr. Wishnu Prasetya

Abstract

Puppet is a language for configuration management that has rapidly gained popularity in recent years. Numerous organizations now rely on Puppet code for deploying their software systems onto cloud infrastructures. In this thesis, we aim to provide a definition of code quality for Puppet code and work towards an automated technique for measuring and rating Puppet code quality. To this end, we first explore the notion of code quality as it applies to Puppet code by performing a survey among Puppet developers. Second, we develop a measurement model for the maintainability aspect of Puppet code quality. To arrive at this measurement model, we derive appropriate quality metrics from our survey results and from existing software quality models. We implement the Puppet code quality model in a software analysis tool. We validate our definition of Puppet code quality and the measurement model by a structured interview with Puppet experts and by comparing the tool results with quality judgments of those experts. The validation shows that the measurement model and tool provide quality judgments of Puppet code that closely match the judgments of experts. Also, the experts deem the model appropriate and usable in practice. The measurement model can provide owners and developers of software systems that include Puppet code with a valuable basis for sustainable software development and effective decision making. The Software Improvement Group has started using the model in its IT management advisory practice.

Contents

Chapter 1

Introduction

In recent years, cloud computing has become increasingly popular [32]. In many cases, deployment of software systems in the cloud entails the use of a configuration management language, such as Chef [26], Ansible [25], or Puppet [27]. These languages allow the software developers and/or system administrators to describe in software what cloud infrastructure an application needs and how the application should be deployed on this infrastructure. Though programs in these configuration languages are executable code, they make use of very different language constructs than normal programming or scripting languages. As a result, practitioners cannot simply apply existing notions of code quality to these new languages. In this thesis, we focus on the Puppet language and explore how software quality can be defined and measured for programs written in Puppet.

In this introductory Chapter, we provide the necessary background on configuration languages in general and the Puppet language specifically. Also, we clearly define the goal of our research, the research questions, and the research method. Finally, we provide an overview of the structure of the remainder of the thesis.

1.1 Configuration management languages

Configuration management languages are used to automatically provision (provide servers with the necessary configuration) and manage servers. The required configuration of a server is written in code, and this gives the nice property that it allows for this code to be re-used as often as is necessary. This allows for the number of servers that are operated to scale at a much higher rate than the required labor to manage them.

Various configuration languages are available with important differences in syntax, programming paradigm, tool support, etc. Whilst Puppet and Chef are Ruby-based, Ansible is based on YAML and Python. Puppet is declarative, which means that you define a final state that the server should be in, and let the Puppet tool figure out how to reach this defined state. Chef and Ansible are imperative, which means that you specify the steps that should be taken to get a host to the required configuration. Various comparisons between languages have been made by others [28, 29, 30]. There is no consensus or established criteria regarding how to choose among languages. Organizations have made choices based on personal preferences, a perceived optimal fit with the problem to be solved, or simply because of past familiarity.

1.2 Puppet

In this thesis we focus on the Puppet language, which is one of the most popular configuration management languages at the moment. Puppet [11] is a Domain-Specific Language (DSL) in Ruby designed to automate server configuration. In Puppet, you can write a configuration script, which describes the state of a machine or multiple machines and this configuration can be executed by Puppet to put the server in the described state.

Bringing the server in the desired configuration can mean different things. With Puppet, you can define which packages should be installed, which services should be running, which users have root permissions, and many more things.

Since Puppet offers a high degree of automation, it is possible to manage a huge number of machines [12]. You only have to write code once, and making use of the automation Puppet offers, apply it thousands of times with only the press of a button.

We will discuss an example of how Puppet can be used to configure servers.

In Puppet, the Puppet Agent application runs on the server, or node. This application is responsible for ensuring that the configuration of the server is correct. This is done by using the Puppet catalog, which is a manifest that contains all resources that should be applied in the final state of the server, and also contains information on the ordering of steps (you cannot start a service without installing it). This catalog is created by a Puppet master, a special server which is responsible for creating the catalogs of Puppet nodes. As a default in Puppet, every 30 minutes the Puppet agent will ask its master for a new catalog, and will then make sure the state of the node is as described in the catalog. This does not mean that everything on the node is defined in the catalog — resources that are not mentioned in the catalog are not managed by puppet. The Puppet master creates the catalog from Puppet code and data from a separate data storage (usually Hiera). The code is written in the Puppet DSL and consists of various modules (a module typically configures a single application). The data in Hiera usually contains data that is specific to a node or company.

We explain the basics behind the Puppet language and show code examples in the next Chapter.

1.3 Why investigate Puppet code quality?

Since configuration management languages such as Puppet allow server configurations to be developed, executed, and maintained just as code in traditional general-purpose languages, it is natural to also look for ways to manage the quality of configuration code. While a lot of work has been done on measuring and managing code quality in general-purpose programming languages, this is not the case for configuration management languages like Puppet, and we do not expect this work to be applicable as-is to these languages.

An example of a quality model to capture code quality is the SIG maintainability model [13, 23]. The SIG model consists of 8 metrics to measure maintainability in software systems, and is language independent. Whilst the SIG maintainability model can be used for Puppet, applying this model to configuration management languages such as Puppet may give problems.

An example of a problem when trying to apply the SIG model to capture Puppet maintainability is to apply the unit complexity metric. In the SIG model [13], unit complexity is the complexity of the units. The definition of a unit is the smallest executable piece of a program. In C# or Java, an unit is a a method. In Puppet, what an unit is, is not clear. It can be argued that the units in Puppet are resources, since these are the smallest executable pieces of code [14]. However in Puppet, it is discouraged to have complexity in resources [15], and it should be outside the resources, in the class. Furthermore, unit testing is done at the class level in Puppet.

If classes were used as units in the SIG model, the unit size metric, would rate any file of 60 lines of code or longer as a very high maintenance risk. In Puppet, having a file of 60 lines or longer is very common and is not perceived as a very high maintainability risk, if perceived as a risk at all.

Puppet also has other properties that are not captured by the current SIG model, for instance how well data is separated from the code. Data in Puppet should go into Hiera, but not all of it — parameter defaults are for instance considered useful to have available right in the code (See ”How do you decide what goes into Hiera and what doesn't?” in Chapter 3). Research on how data should be separated in Puppet would be very applicable to other configuration management languages that also have a comparable data storage.

The SIG model was developed to be as language independent as possible. It has been applied successfully to a wide range of programming languages, including procedural and object-oriented general-purpose programming languages. We think that the model can be a good basis also for configuration languages, but an adapted version will be needed to do justice to the differences with traditional programming languages.

Investigating Puppet is also relevant since it is an up-and-coming technology, having experienced large growth in the past few years. Since there is no good quality model for Puppet, it would be very useful to have one at this point in time, to make sure that software-defined infrastructures that already exist or are in the process of being created are also future-proof. For these reasons, we are going to investigate Puppet code quality.

1.4 Research Questions

The goal of our research is to discover how to best measure code quality in Puppet configurations. We consider configurations to be entire collections of modules that work together to form a software defined infrastructure. For this we want to measure both code quality at the file level and quality on a more abstract level, such as the architecture and other configuration-wide aspects.

The research questions are as follows:

Research Question 1: What quality aspects of Puppet code quality can be measured?

RQ 1.1 What quality aspects of Puppet code quality can be measured at the file level?

RQ 1.2 What quality aspects of Puppet code quality can be measured at the module / configuration level?

Research Question 2: How can these quality measurements be combined into a measuring instrument?

Research Question 3: Are the quality measurements from RQ2 a good indication of Puppet code quality?

RQ 3.1 Is the combination of measurements from RQ2 minimal and complete?

RQ 3.2 Is the combination of measurements from RQ2 considered useful in practice by Puppet experts?

1.5 Research Method

To obtain answers to the research questions above, we following the following approach.

1. We develop and run a survey among Puppet developers to better understand what current understanding they have of code quality issues in Puppet code and how they currently deal with those.
2. We collect possible quality metrics for Puppet code from survey answers, from existing quality models and tools, and from literature.
3. We construct a quality model that aggregates selected quality metrics into a coherent rating model
4. We implement the rating model and apply it to a wide range of Puppet code bases
5. We validate the rating model against the quality judgments of Puppet experts

1.6 Structure of Thesis

In Chapter 2, we will give a short introduction to Puppet and discuss a few code examples. In Chapter 3, we will discuss our survey which serves as the basis for Chapter 4. In Chapter 4, we will explain the metrics that we have selected to measure in Puppet, and answer research question 1. In Chapter 5, we will prune the set of metrics from Chapter 4 to arrive at our quality model, and answer research question 2. In Chapter 6, we explain how we measure Puppet code quality. In Chapter 6 we discuss how we implemented the metrics of the model to arrive at a quality judgment. In Chapter 7 we will explain how we calibrated our measurement model, and how we rated configurations. Chapter 8 describes the validation of our model, which consists of interviews with Puppet experts and this is the answer to research question 3. We discuss related work in Chapter 9, and our conclusion is in chapter 10.

Chapter 2

Puppet

In this Chapter, we will give a brief introduction to the Puppet language, in order to explain language concepts discussed in further Chapters. We explain what resources, classes, modules, facts and nodes are, and also explain how Puppet works with Ruby.

2.1 Resource

In Puppet, the basic building block is called a resource. Each resource has a resource type. Examples of commonly used resource types are the file, package and service resource types. There are 48 predefined resource types, but additional custom resource types can be added.

```
package { 'openssl':
        ensure  => installed
}
```

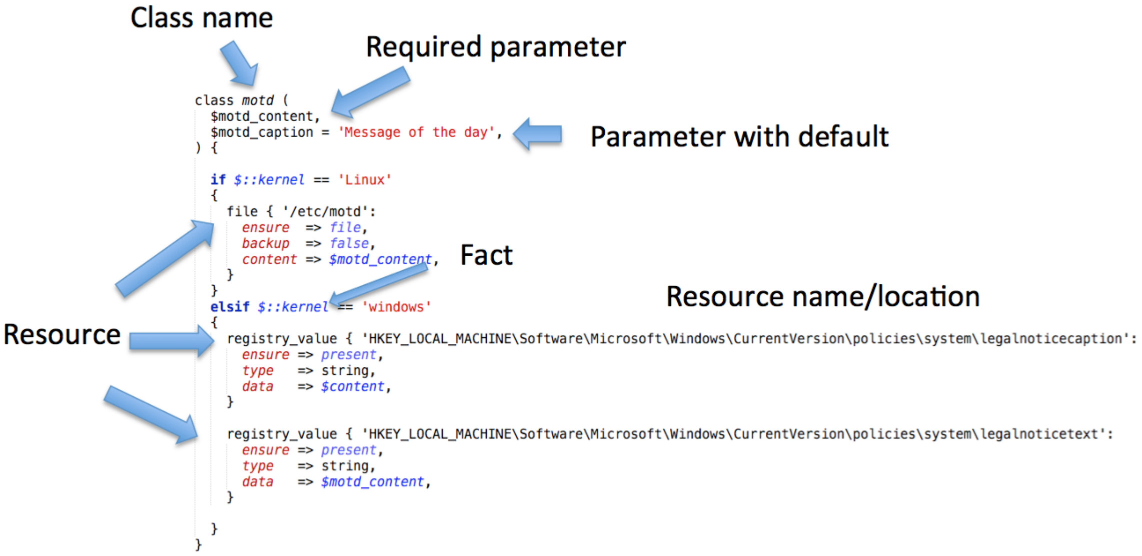
Listing 2.1: Example of a Puppet resource. This resource tells Puppet that the openssl package should be installed (if this has not been done not already). This resource is of the resource type package, and this resource type is predefined in Puppet.

2.2 Class

Classes in Puppet are different from classes in normal object-oriented programming languages. In Puppet, a class is most importantly a collection of resources.

To give an example of a class in Puppet, we modified the Puppetlabs motd module [33] and annotated it (see figure 2.1).

The task of this class is to configure the message of the day for a system. It has two parameters. *$motd_content* is a required parameter (the value always has to be provided), and *$motd_caption* is a parameter with a default value (this means that a value does not always have to be provided). To use this class, the value for $motd_content has to be provided, the one with a default does not have to be provided — unless the user wants to provide a different value. Three resources are defined in this example. The file is for linux-based systems, and sets the message of the day for those systems. For windows, two registry values have to be edited.



```
class motd (
  $motd_content,
  $motd_caption = 'Message of the day',
) {

  if $::kernel == 'Linux'
  {
    file { '/etc/motd':
      ensure  => file,
      backup  => false,
      content => $motd_content,
    }
  }
  elsif $::kernel == 'windows'
  {
    registry_value { 'HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\policies\system\legalnoticecaption':
      ensure => present,
      type   => string,
      data   => $content,
    }

    registry_value { 'HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\policies\system\legalnoticetext':
      ensure => present,
      type   => string,
      data   => $motd_content,
    }

  }
}
```

Figure 2.1: An example of a class in Puppet.

2.3 Facts

Facts in Puppet are system-specific variables collected by the Facter tool [8]. Facts are variables that can be used in Puppet code. Puppet supports built-in facts as well as custom facts.

Built-in facts in Puppet offer a powerful tool to get system information, since an implementation is already provided for the user. In generic facts, that are not OS-specific, this implementation is provided for Solaris, Linux, Windows and Mac OS. Examples of OS-specific facts, where an implementation exists only for a single platform, are *macosx_buildversion* and *solaris_zones*.

In figure 2.1, the $::kernel fact is there to check what kind of operating system kernel is used. In this case, $::kernel is a built-in fact in Puppet. Puppet maps the name of the fact to the specific implementation on the system which it operates on.

2.4 Exec

Another example of a predefined resource type in Puppet is an exec. An exec is a shell command in Puppet. Execs play a large role in the next Chapters of this thesis, so we explain it here.

An example of an exec is the following:

```
exec { 'cabal update' :
          user         => $user,
          environment  => "HOME=$home",
          path         => ['/bin', '/usr/bin', '/usr/local/bin'],
}
```

Listing 2.2: The shell command to update cabal (a package manager for Haskell), as an exec in Puppet.

In this example it is easy to figure out what happens (the Cabal package manager for Haskell updates the package list), but there are some downsides to having an exec. You have to read the command to figure out what it does (this can get difficult when the shell command is complicated). There is also no stop-condition in this exec — it will run every time the Puppet agent runs. In this case it is harmless, but in other cases this may have unintended side-effects.

2.5 Module

The above example consists of only a single file. In Puppet, you write modules, typically to configure a single application or service. A module consists of one or multiple files to configure the service, and it may consist of: tests (typically unit tests with rspec, or/and acceptance tests with beaker or serverspec), a readme, templates in embedded Ruby, custom types and providers in Puppet, files, example code, and custom facts.

Templates in Puppet are Embedded Ruby or Embedded Puppet files, that can be rendered by providing parameters to them. This is useful when a configuration file is dependent on variables, such as the number of cores or the amount of available memory. A template can be used to abstract over these variables, instead of requiring a separate file for every use case.

Files in a Puppet module are files that the module needs to function, unlike templates, these cannot be edited by providing parameters.

Custom types and providers are custom resources, written in Ruby. The type provides the interface, such as the name of the resource and what attributes it has, whilst the provider contains the implementation.

Custom facts are extensions to built-in facts in Facter. They can be useful when a module requires specific information from a node that built-in facts do not provide.

2.6 Node

Whilst resources, classes and modules are the tools used to get the job done, Puppet runs on nodes (servers), and to tell Puppet what kind of catalog (which describes the desired state of a node) should be applied to a node, the node definition must be given.

An example node definition is the following:

```
node 'test' {
        class { 'motd' :
                motd_content => 'Welcome to Chapter 2!'
        }
}
```

Listing 2.3: Example of a Puppet node definition.

In this very simple example, only motd is used. If the motd class would not have had required parameters, it could have looked like this:

```
node 'test' {
        include motd
}
```

Listing 2.4: Another example of a Puppet node definition, in this case for the motd class if it did not have required parameters.

2.7 Further Reading

We provided only a brief introduction. Many concepts we mentioned or touched upon are elaborated further in the Puppet documentation [31]. There are other books available about Puppet, but we consider this source to be the most up-to-date with recent language concepts, whilst also being accessible to those unfamiliar with Puppet.

Chapter 3

A survey on code quality among Puppet developers

In this Chapter, we answer research question 1 by doing a survey on Puppet code quality among the GitHub population of Puppet programmers. We explain how we selected our questions, survey population, and we discuss the results.

In Chapter 1 we already explained that we think there is a better model possible to capture Puppet code quality. Whilst we have some general knowledge on maintainability and Puppet, this knowledge might be different from other Puppet experts. We decided to hold a survey to get a better idea of Puppet code quality, as well as find additional measurements and aspects for our model.

3.1 Survey: Population

The intended target of our survey are Puppet developers. We tap into that community through GitHub, the currently most popular web-based code repository service. We have carefully designed our sampling method to ensure a sample that is of sufficient size and includes Puppet developers with a sufficient level of experience.

To determine the survey population, we used the GitHub API to find all users that had more than 10 commits in at least 2 repositories each that were labeled by the GitHub API as containing Puppet, and that were pushed in the last six months. In addition to this, we also required the text 'Puppet' to appear in the repository clone URL, since some repositories were simply copies of a Puppet/virtual box template, without any Puppet programming being done.

To establish this, we joined the table of contributors and users to add email addresses to the result (Users without emails cannot be reached, so they are excluded). We joined this table with the table of repositories, and then filtered out all contributors with 10 or less commits. Then, every contributor that appeared more than once was selected for the survey population.

3.2 Survey: Questions

Given the absence of established theory or literature on Puppet code quality, we decided to aim for a survey that is explorative in nature. For this, we opted for open questions, rather than closed ones, and we chose to reach out with a short list of question to a large audience, rather than aiming for an in-depth survey which only a limited number of developers would be willing to participate in. We created an initial survey with which we carried out a pilot. The results led us to make considerable adjustments to the survey. This new survey was piloted twice before proceeding to carry out the survey at scale.

To determine the questions of the survey, a large list of candidate questions that might be relevant was created. Then, we discussed the questions with various experts — on areas of software engineering, empirical studies, and Puppet — and we removed questions that would invite vague answers, questions that were difficult to understand or questions whose answers would be irrelevant to our research. This reduced the survey to about 10 open questions.

Our initial survey was generic. Some example questions are ”When do you refactor Puppet code?” or ”What are typical issues you encounter when reading Puppet code?”. Whilst people familiar with maintainability gave useful answers in our internal pilot of 5 Puppet programmers, external respondents (19 people in the first pilot) did not give answers we considered useful.

We decided to change our approach, and ask questions related to quality — what do people look for when considering quality, or what are examples of bad practices. To get around the problem of having to pilot with external Puppet experts (depleting the useable population), we instead sent the survey to programmers not familiar with puppet, and asked them to pretend it was their favorite programming language. The results from this were considered useful — the answers were mostly related to quality and more importantly, the answers consisted of quality aspects that we could actually measure. We piloted externally for a second time, and after removing one question from this survey, we arrived at the actual survey we would use.

In tables 3.1 and 3.2, we show the questions for pilot 1 and pilot 2 of the survey. In addition to this, we provide reasons for removing or keeping the questions for further surveys.

Question	**Assessment**	**Scrapped**
How many years of experience do you have with programming puppet?	Useful question to chart population.	No
Which issues do you typically encounter when reading puppet code?	Respondents interpreted the question in different ways.	Yes
What aspects do you take into account when selecting a puppet module from GitHub or Puppet Forge?	Different and interesting responses.	No
Do you typically re-use puppet code, across modules or deployments? How?	Respondents indicated that they re-use code when possible.	Yes
When you are writing a module, when and how do you divide functionality between classes?	Respondents interpreted the question in different ways.	Yes
How do you test your puppet code?	All respondents tested their code using testing strategies that are commonly known.	Yes
Do you use Hiera? How do you decide what goes into Hiera and what doesn't?	All respondents used Hiera, but they used it in different ways.	No
How do you recover from errors in an applied configuration?	"Fix the errors" was the most common answer.	Yes
When programming puppet, what activity takes up most of your time?	People indicated what activity took up most of their time, but this response was less promising than expected.	Yes
What is your biggest problem when programming puppet?	This question had interesting responses, not only for our research, but also for others that are interested in Puppet.	No

Table 3.1: Survey Pilot 1. We show the questions we asked, our quality judgment to scrap or keep the question, and a reasoning behind it.

Question	**Assessment**	**Scrapped**
How many years of experience do you have with programming puppet?	Same as in table 3.1	No
How large is your puppet setup? (#Modules, #LOC Puppet, #resources, #nodes, Lines of Code)	Some parts of this question were not answered in our pilot. This might have been because it was too difficult (we could have provided a bash command) or because respondents did not want to answer this.	Yes
If you had to judge a piece of puppet code (file or module) on its quality what would you look for?	The answers to this question were useful. It was easy to create metrics from the responses.	No
Could you give some examples of bad practices in puppet code?	Useful answers were given that were also possible to measure in a quality model.	No
What aspects do you take into account when selecting a puppet module from GitHub or Puppet Forge?	Same as in table 3.1	No
How do you decide what goes into Hiera and what doesn't?	Same as in table 3.1	No
What is your biggest problem when programming puppet?	Same as in table 3.1	No

Table 3.2: Pilot Survey 2

Our actual survey consisted of the following questions:

- How many years of experience do you have with programming puppet?
- If you had to judge a piece of puppet code (file or module) on its quality what would you look for?
- Could you give some examples of bad practices in puppet code?
- What aspects do you take into account when selecting a puppet module from GitHub or Puppet Forge?
- How do you decide what goes into Hiera and what doesn't?
- What is your biggest problem when programming puppet?

3.3 Survey: Execution

We sent out our survey to 257 Puppet programmers, and got 40 responses. We added 2 from the previous pilot to the analysis, since the same questions were answered in this pilot as in the actual survey.

Survey	Population size	Delivery notification failures	Responses	Response rate
Pilot 1	19	1	3	16,6%
Pilot 2	20	0	2	10%
Actual Survey	257	5	40	15,8%

Table 3.3: Overview of the survey populations and response rates.

3.4 Survey: Analysis

We analyzed the survey by coding the answers to the open questions, similar to Gousios et al. [17]. We required that the response that was given answers the question. We required at least two responses for a code where possible, and tried to group unique responses together in a more general category. Since some answers consisted of long enumerations of quality aspects, we used no upper bound on the number of codes a response could get.

The initial results of the survey have been published online [18].

3.5 Survey: Results

In this section, we briefly discuss the questions of our survey. The first question is about programming experience, and the answers to this question are shown in the bar chart in the next paragraph. We discuss questions 2 to 6 by summarizing the analyzed data. The graphs showing the codes and their frequencies are included in appendix A. The dataset and code table are available online [19].

How many years of experience do you have programming in Puppet?

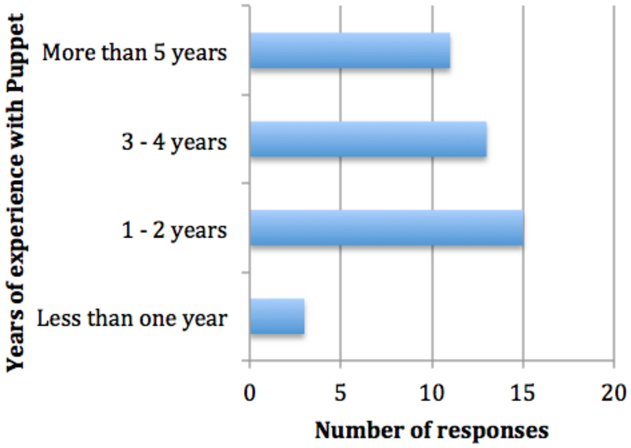


Figure 3.1: A bar chart showing the answers to question 1.

The responses to this question are visible in Figure 3.1. Our population ranged from new developers of Puppet, to very experienced Puppet developers.

If you had to judge a piece of Puppet code (file or module) on its quality, what would you look for?

Looking at the top 5, code quality in Puppet is not very different from normal programming languages. Testing, documentation, code style and linting are common concepts in general purpose programming languages as well. More interesting is the program structure — examples here are proper usage of the Package, Config, Service design pattern, and making use of a params class to set defaults. The exec resource is also frequently mentioned and specific to Puppet. Also mentioned in the answers is that the usage of exec should be kept to a minimum.

From the infrequently mentioned categories, the design, readability, complexity and dependencies are also common quality factors in general purpose programming languages. Puppet-specific categories are parameters (how well is the module parameterized, what does it expose?), custom types, being cross-platform and behavior.

Respondent 7 answered: ”Readability, adherence to style guides, appropriate organization of code into classes and modules, use of a params class, separation of data from code, sparing and responsible use of exec resources, good comments and documentation, good test coverage. See also: https://forge.puppetlabs.com/approved/criteria”

Could you give some examples of bad practices in Puppet code?

The leading bad practice here is the improper usage of exec. Some other Puppet-specific bad practices are a lack of program structure (no PCS / params.pp for instance, or having very large files), hardcoded variables (that should have been parameters), having too many dependencies (or too few), non-idempotence (a catalog applied multiple times should not result in changes being made every run) and having too many or too few parameters. Also failing to use Hiera when this is needed is considered a bad practice.

Aside from that, not having documentation, having a very complicated program or failing to test are also examples of bad practices.

Examples of bad practices according to respondent 11: ”Using exec resources to shell out to custom scripts unnecessarily. Bad indentation. Non-idempotent code. Failure to use params.pp or Hiera when appropriate. Lots of static variables set in code when they should be set as class parameters.”

What aspects do you take into account when selecting a Puppet module from GitHub or Puppet Forge?

The answers to this question are not specifically related to Puppet code quality, and are more an example of how people use third-party code. Only four respondents indicated that they looked at the code.

The activity of the repository, the documentation, whether other people use it, who the author is (and his/her reputation), and the offered functionality of the module are the most frequently used categories.

Different views exist on the what people look for. Respondent 21 takes the following into account: ”Rate of maintenance, code quality in github repo, who made it, what version it is, is it for my OS and how many people seem to be using it?”. Respondent 4 looked for different aspects, and required that using the third-party module significantly reduces the programming burden: ”Whether it’s clear from the documentation that the author understood cross-platform programming and the underlying service itself sufficiently that using the module would be significantly less work than writing our own.”

How do you decide what goes into Hiera and what doesn’t?

The codes for this answer differ from the other questions. There are more codes and the responses per code are lower – there is a number codes that have only one response.

It appears from this data that people use Hiera in different ways – most responses indicated that data is stored in Hiera and code is in Puppet, but it appears there is a difference in opinion on what is data and code. Some people put defaults in Puppet, but another indicated that he puts them in Hiera (module-data). A recurring pattern seems to be varying data, such as data that is company, environment or site specific should go into Hiera.

In other cases, respondents mention that they put certain data in Hiera, but others do not confirm or deny that view. Operating System-specific information should not go in Hiera. An example of such a variable is the package name for the Apache application. In Debian, it is simply called apache, but in RedHat it is called httpd. The best practice is to have these differences in the code, and not in Hiera.

Respondent 23 explains his point of view on data and code separation: "All variable data (user names, file and directory names, etc.) goes into Hiera. Where it makes sense a class may contain sensible defaults for parameters or even logic for computing a value in the absence of a parameter."

What is your biggest problem when programming Puppet?

The responses to this question were mostly unique – different respondents have different problems in Puppet. Problems were grouped together based on their commonalities.

Most reported problems are related to the way that Puppet is designed. Examples are snowflake servers, the limitations on a single node, and nondeterminism.

Problems related to third-party modules are also a common theme: "We don't have many problems. The biggest problem we've run into is relying on a third-party module from Puppet forge or Puppet-labs. We've eventually had to either: — make forks of some modules to include some of the features we need (or fix a bug) — switch to another module". The poor quality of third-party modules was also mentioned.

Problems with testing also appeared, especially writing good quality tests.

Other examples of problems are the documentation of the types and providers API, functionality that people wanted in Puppet but Puppet does not have, and the quality of error messages – both Puppet and Ruby-based.

Finally, a few respondents mentioned a lack of best practices. One respondent mentioned a lack of best practices for the toolset, another mentioned a lack of a best practice for module organization – the modules folder can get large, sometimes over 200 modules – can this be organized better, and how?

Chapter 4

Selected Puppet code metrics

In this Chapter, we will explain what metrics we will use to measure the quality/maintainability of Puppet, and answer research question 2. The metrics that we will measure are either from our survey, or from the SIG quality model. Since the performance of these metrics is not yet analyzed, the quality model will be presented in Chapter 5 after analyzing the initial performance of the metrics.

4.1 Selection Criteria

We used the following criteria to select our measurements:

- Easy to measure
- Only measures Puppet source code
- Metric is applicable on almost all codebases
- Metric is promising w.r.t. quality / maintainability
- There is a clear opinion on what is good quality for the metric

We have opted for metrics that are easy to measure. Some things are difficult to measure, such as the quality of the documentation or test quality. We have restricted ourselves to only measuring Puppet source code. We want measures that are applicable across almost all codebases. If 90% of the codebases doesn't have a certain problem, rating a score from 0.5 to 5.5 on a certain aspect will give a large number of 5.5 stars, and this can inflate the rating. Finally, we want to have metrics that seem promising w.r.t. quality / maintainability. If a metric is easy to measure but does not tell you much about the quality or maintainability, or it would be very hard to derive a quality judgment from it, we have opted not to include them in our measurements. Finally, we demand that there is no possible conflict between scoring good or bad on a metric and the opinion of good quality and low quality. This is especially true for the separation of data / code. From the survey, we can see that there are different or conflicting opinions.

4.2 Metrics from Survey

We have selected the following metrics from the survey to measure:

File Length

‘Single Class Shabang’ was mentioned in the survey, as well as the number of lines. Having too much code inside a single file (which according to best practices, should only contain one class [4]) is an indication that too much is going on in the file to understand it.

Complexity

Complexity, simplicity and if-statements were mentioned in the survey results, so we will measure the complexity. This is measured at the file level.

Number of exec statements

The number of exec resources was mentioned in the survey, and the best practice is to minimize this number. An exec is a shell command in Puppet, and has been previously discussed in Chapter 2. We asked why execs were bad in our validation interviews in Chapter 8.

Puppet-lint

Puppet-lint [10] is a tool that is used to check if Puppet code adheres to the Puppet style guide. This concerns both style elements, such as detecting the use of tab characters, but also checks for things that will most likely introduce errors in your code, such as quoted booleans. Adherence to the style guide and Puppet-lint were mentioned frequently in the survey, so measuring the number of lint warnings and errors is a possible quality measure.

Number of resources

Resources are the basic building blocks of Puppet, so measuring resources could be an interesting quality measure to detect whether a file has too many responsibilities. Since the ‘best practice’ with execs is to use the proper (custom) resource, we also measure the number of custom resources.

References

Puppet offers many different types of references. Some are across different classes, such as a class including or referencing another class. Other references exist between resources, such as a resource with a metaparameter requiring it to be applied after a certain other resource. References can be between files or internally in files. We measure both fan-in from files (references to this file), fan-out from files (references to another file) and internal references (references from within the same file).

Relations between resources

We also measure the relations between resources. This can either be arrows or metaparameters. In Puppet, these specify ordering, i.e. this resource should be applied before that one.

Parameters and variables

Parameters and variables were mentioned in the survey. Since it is difficult to measure whether a variable or parameter is useful, we have decided to measure how many variables / parameters per file are present. Since parameters can have defaults, we also measure how many required parameters there are: parameters where a value must be supplied when calling the class.

Hardcodes

Any hardcoded value is considered a hardcode — such as ’root’ — which might be harmless, whilst a hardcoded ip-address, email, password, certificate or private key in the code is probably bad. Magic constants are also a common code smell, thus we measure the number of hardcodes.

Other metrics

Other metrics that we looked for were the number of arrows, global defaults, classes per file, internal calls, and the number of required parameters of a class/define. Finally, as a measure to rate the architecture of a Puppet configuration, we measure the degree (all incoming and outgoing calls) of the modules.

4.3 Metrics from other models

From the SIG model [23], we borrow volume, duplication and file-level fan-in.

Volume

We measure volume, since the larger a codebase is, the more maintenance work we can expect to be done. It is also a quality aspect — if two codebases are very different in size, but accomplish the same thing, than the smaller one could be considered of higher quality since it needs less code to do the same job.

Duplication

The disadvantages of duplication as listed in Building Maintainable Software [23] are that changes that have to be made in a duplicated block of code, also have to be made in the duplicates of this block. This is both more maintenance work and can lead to problems when other duplicates are not updated — or when it becomes very difficult to identify other duplicates. Duplication in code is bad and should be measured.

Fan-in

The reasoning behind measuring fan-in is also explained in Building Maintainable Software. It is possible that ripple effects are created when editing tightly coupled files, which makes it difficult for maintenance work to take place. An additional advantage is that the codebase becomes easier to navigate. There should be no single class or module that does ’everything’, and responsibilities should be located in the appropriate file or module.

The above metrics in the SIG model are benchmarked across technology. For our model we want to compare Puppet codebases, so we have decided to create new thresholds for the metrics in the SIG model, instead of reusing existing ones. A very large Puppet codebase might seem very small when compared to Java, so deriving new thresholds avoids that we give all Puppet codebase very high ratings on volume (and thus skew our comparison). On the other hand, it might be possible that Puppet suffers from more code duplication than other languages, so we also create new thresholds for that. We also look at file-level fan-in (module coupling in the SIG model). Again we will look for new thresholds.

The SIG model has other metrics, such as unit interfacing, unit complexity or unit size. Whilst the above metrics can be considered similar — we look at filesize, number of parameters or the complexity of the file, the level at which we measure is different. They are also separately mentioned in the survey, which is where we got these metrics from originally.

4.4 Metrics that we did not select

We give a short, but by no means exhaustive, list of metrics that we did not measure and we explain why

Package, Config, Service-pattern

The Package, Config, Service-pattern [2] was mentioned often, and it concerns how responsibilities are setup in a module. We consider it a programming convention. It might be the case that you don't use it your code is still maintainable — it might also be the case that you use a slightly different convention, in which case you would get punished. That being said, the Package, Config, Service-pattern was remarked in our validation interview (number 7) as being a plus.

Location of platform complexity

Platform complexity is induced by a conditional or other control statement which is dependent on an OS-variable, such as whether the node is on Debian or Windows. It can be a challenge to figure out when platform complexity occurs. The params.pp pattern states that this should be in the params.pp file. This is also a convention, and sometimes the complexity value can become so large that a metric to measure the usage of params.pp and the file length start to conflict (an increase in one of the ratings will lead to a decrease in the other). It can also conflict with the metric that states that files should not be very complex.

Ruby

Some people look at the amount of in-line Ruby in the Puppet code. Inline Ruby is deprecated in Puppet 4, so measuring how much you have and associating a quality score to it is not necessary — the amount of in-line Ruby should be zero, and it is not necessary to make a comparison. It should be phased out so you can upgrade to Puppet 4, if this has not been done already.

Exotic language features

Puppet has some exotic language features, such as the '+>' operator which adds elements to an inherited resource, or the iteration operator. Whilst some people considered them quality aspects, the '+>' is considered to be very rare, and iteration only happens in Puppet 4, which means that only in the very rare use case that it is used inappropriately this would become an issue. Thus, we did not measure this.

4.5 Answer to Research Question 1

Research question 1 was: *What quality aspects of Puppet code quality can be measured?* This question was split in two sub-questions:

RQ 1.1 What quality aspects of Puppet code quality can be measured at the file level?

RQ 1.2 What quality aspects of Puppet code quality can be measured at the module / configuration level?

We have decided on the metrics listed in table 4.1 that we will measure. We consider volume, duplication, module degree and the fan-in of files (besides internal calls) to be metrics that try to measure quality on a more abstract (module/configuration) level, and the other metrics are measured on the file level. We analyze the usefulness of these metrics in the next Chapter, and will also present our quality model there.

Metric
Filelength
Number of resources
Number of exec resources
Complexity
Number of warnings
Number of errors
Number of parameters
Number of variables
Number of arrows
Number of global defaults
Number of classes
Number of hardcodes
Number of required parameters
Number of outgoing references (fan-out)
Number of incoming references (fan-in)
Number of internal references
Module Degree
Volume (Lines of Puppet) of the configuration
Duplication (percentage of redundant lines)

Table 4.1: This table lists the metrics that we will measure.

Chapter 5

Constructing the quality model

In the previous Chapter, we have presented candidate metrics to measure. In this Chapter, we will study the metrics empirically and further refine the set of candidate metrics, until we arrive at a set of metrics that is suitable for our quality model. We will do this by gathering Puppet repositories from GitHub, and see how the file-level metrics perform, and prune this set by removing unnecessary metrics. This will result in the metrics of our quality model. In addition to this, we also explain the rationale behind these metrics — what they should measure and why this is important from a quality perspective.

5.1 Data collection

Using the GitHub search API we collected the clone URLs of all Puppet repositories on GitHub, the number of which was 16.139 at the time of collecting. On February 1, 2016, we cloned all Puppet repositories, except repositories that were removed or inaccessible. We used shallow cloning and cloned recursively where possible — some submodules were in private repositories and could not be reached.

5.1.1 Dataset cleaning

The GitHub dataset contained repositories that were marked as Puppet code (because of the .pp extension), but that were not actually Puppet code. We encountered Pascal, assembler, bytecode, PowerPoint, a dialect of SQL, preprocessor files [7] and various other languages. This was discovered because our extended Puppet-lint parser assumes that some Puppet language constructs exist, or because the Puppet-lint parser failed to parse the file. All files that gave parse errors were removed from our dataset. Our cleaned dataset consisted of 15.540 repositories with Puppetcode. The difference from the earlier 16.139 is both because we removed code with errors, we did not analyze code that was stored inside hidden directories, and some repositories were removed by GitHub users between collecting the repository URLs and cloning the repositories.

5.2 Usefulness of Lint Warnings and Errors

Figure 5.1 is a grouping of all the lint warnings and errors in the full dataset from GitHub. As is clearly visible, spacing and quoting errors and warnings are far more frequent than others.

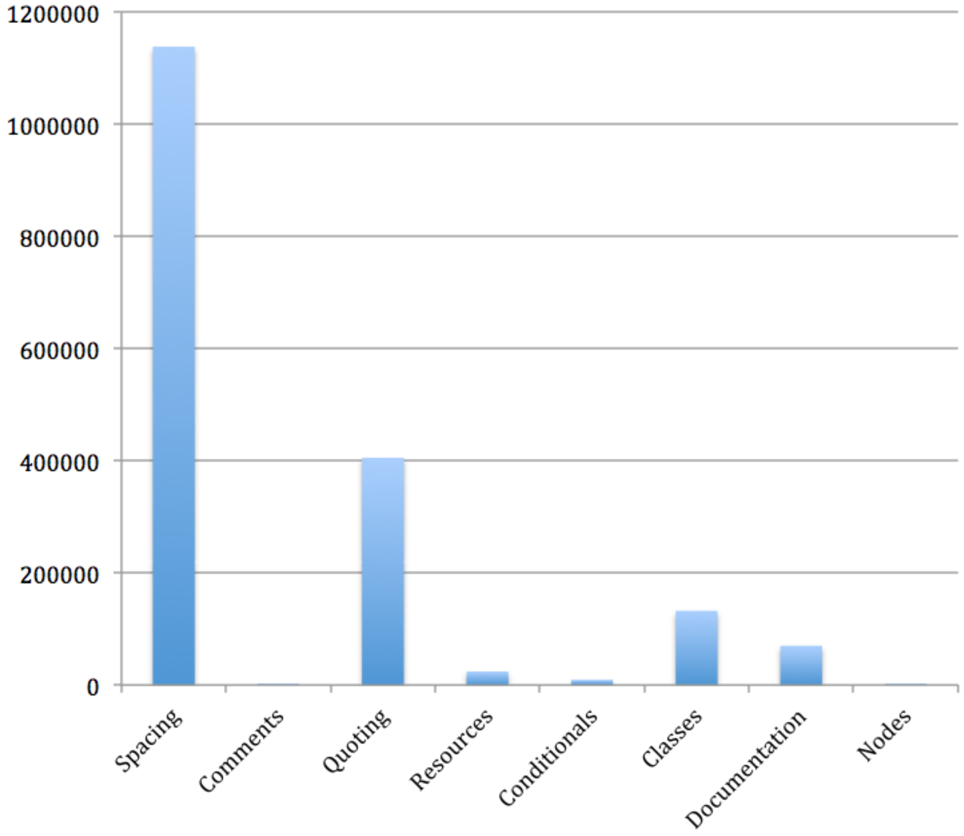


Figure 5.1: The number of Lint warnings and errors, grouped by the category of the warning/error, for the entire dataset we collected from GitHub.

We made a selection of the warnings and errors and opted to only take warnings and errors that not simply style violations account. The errors and warnings we selected are shown in table 5.1 and table 5.2.

single quoted string containing a variable found
”foo::bar not in autoload module layout”

Table 5.1: We measured only the lint errors listed in this table.

quoted boolean value found
string containing only a variable
variable not enclosed in
ensure found on line but it's not the first attribute
mode should be represented as a 4 digit octal value or symbolic mode
unquoted resource title
case statement without a default case
selector inside resource block
not in autoload module layout
class defined inside a class
class inherits across module namespaces
right-to-left (<-) relationship
top-scope variable being used without an explicit namespace
class not documented
defined type not documented

Table 5.2: We measured only the lint warnings listed in this table.

This selection is done so that lint warnings and errors become a measurement of quality, instead of the adherence to a style guide. In the case of filtered lint errors, the number of errors you could get in a single file was so low that it was impossible to derive a quality judgment from them. We thus opted only to use filtered lint warnings in our quality model.

5.3 Pruning the metrics

The list of metrics in table 4.1 is long, and we have no idea how well these metrics might perform. We used statistical analysis on the dataset to see what metrics are correlated, and what metrics might be good candidates to measure or that might have to be removed. Metrics that are closely correlated to other metrics, metrics that don't really say anything about quality, and metrics that are only present in a very small amount of code, are not useful to measure.

To assess which metrics are correlated, we listed all the metrics for each configuration of our benchmark set (see section 7.1) and used Spearman's correlation to see which metrics are correlated.

From table 5.3, it is clear that both the number of resources and the number of hardcoded values (which does not cover the content of the hardcodes) is strongly correlated with filelength. Therefore, we have opted not to include resources into the model. We excluded the number of hardcoded values as well. In addition to this, different users have different policies on what data should go into Hiera, and what should go into code, so enforcing a single standard in the model might not work very well when it is used in practice.

	filelength	#resources	#exec	complexity	#warnings	#errors	#parameters	#arrows
filelength		0.80	0.29	0.52	0.25	0.22	0.52	0.25
#resources	0.80		0.29	0.33	0.21	0.25	0.32	0.21
#exec	0.29	0.29		0.21	0.14	0.09	0.18	0.13
complexity	0.52	0.33	0.21		0.24	0.14	0.40	0.22
#warnings	0.25	0.21	0.14	0.24		0.31	0.05	0.16
#errors	0.22	0.25	0.09	0.14	0.31		NA	NA
#parameters	0.52	0.32	0.18	0.40	0.05	NA		0.18
#arrows	0.25	0.21	0.13	0.22	0.16	NA	0.18	
#globaldefaults	0.06	0.04	NA	0.06	NA	0.05	NA	0.09
#classesperfile	0.30	0.32	0.13	0.20	0.30	0.62	0.17	NA
#hardcodes	0.90	0.81	0.25	0.40	0.18	0.22	0.34	0.20
#requiredparams	0.26	0.16	0.11	0.17	0.10	0.05	0.58	0.10
fanout	0.43	0.44	NA	0.14	0.07	0.17	0.18	0.11
fanin	0.23	0.15	0.14	0.21	0.09	-0.06	0.16	0.09
internal	0.52	0.56	0.37	0.23	0.13	0.25	0.25	NA
	#globaldefaults	#classesperfile	#hardcodes	#requiredparams	fanout	fanin	internal	
filelength	0.06	0.30	0.90	0.26	0.43	0.23	0.5	
#resources	0.04	0.32	0.81	0.16	0.44	0.15	0.6	
#exec	NA	0.13	0.25	0.11	NA	0.14	0.4	
complexity	0.06	0.20	0.40	0.17	0.14	0.21	0.2	
#warnings	NA	0.30	0.18	0.10	0.07	0.09	0.1	
#errors	0.05	0.62	0.22	0.05	0.17	-0.06	0.2	
#parameters	NA	0.17	0.34	0.58	0.18	0.16	0.3	
#arrows	0.09	NA	0.20	0.10	0.11	0.09	NA	
#globaldefaults		-0.05	0.06	0.00	0.00	NA	NA	
#classesperfile	-0.05		0.27	0.14	0.12	0.17	0.3	
#hardcodes	0.06	0.27		0.11	0.40	0.16	0.5	
#requiredparams	0.00	0.14	0.11		0.10	0.12	0.1	
fanout	0.00	0.12	0.40	0.10		NA	0.2	
fanin	NA	0.17	0.16	0.12	NA		0.2	
internal	NA	0.31	0.51	0.12	0.17	0.18		

Table 5.3: Spearman correlations of listed metrics. Correlations with P-values higher than 0.01 are listed as NA.

The mean of the number of lint errors per file (across all files in the dataset) (0.4), arrows (0.3), and global defaults (0.04) was very low, so we decided not to include these metrics in our quality model.

Defining Global Defaults is bad inside modules, but in the benchmark set this happened only inside the main manifest, where this is not an issue (an interviewee during the validation session remarked this as well). In a rare case the number of arrows is very large, like over 200, but usually it is small, and thus not useful as a metric to measure across all configurations.

We opted to use file fan-in instead of other metrics such as fan-out, the number of metaparameters, and the number of internal calls. The reason for this is based upon the reasoning used in the SIG model [23] (see also section 4.3). High fan-out, a large number of metaparameters, and internal calls might mean that the file itself is already too large, so we consider measuring the filelength an adequate measure. The fan-out of modules is measured as part of the module degree metric.

The mean of the number of classes per file is 1.2 in our benchmark (1 configuration had all code inside a single file), so it might be useful to flag this as a violation in the model to enforce 1 class per file [4].

In addition, we tried some metrics calculating the density of some things (such as resource or exec density), but this proved to be very strongly correlated to the metric already calculated. An example of such a density metric is exec density. We calculated both the number of execs divided by the number of resources, and the number of execs divided by lines of code per file. The former had a Spearman correlation of 0.989 with the number of execs, and the latter had a correlation of 0.990 with the number of execs. Both had p-values below $2 * 10^{-6}$.

Finally, we have opted to exclude the number of variables from the model. We do not consider it a great metric for multiple reasons. First, if we plot the quantiles of how variables are distributed across files in our benchmark set (see table 5.4), they are not distributed across all the files — the majority of the files do not have variables. Second, we are not sure what variables actually measure. If a file has too many variables, we expect that it is already too large and that filelength should cover this. If a file has not enough variables (as suggested in validation interview 9), some context is needed to determine that variables are not used correctly, and the number of variables is not an adequate metric to measure this.

10%	20%	30%	40%	50%	60%	70%	80%	90%	95%	100%
0	0	0	0	0	0	1	2	4	9	838

Table 5.4: Quantiles of the number variables per file

5.4 Quality Model Metrics

After filtering the above metrics, we arrived at the following list of metrics that we think is useful to measure. The reasoning for the metrics has been explained in Chapter 4, here we explain the implementation.

Filelength

We count the number of total lines of code in a file. Lines containing only commentary, whitespace or braces are ignored.

Complexity

To measure complexity, we add together the number of if-statements, the number of selectors, the length of the cases of the case statements and the number one (the base branch). This is similar to McCabe, although we do not fully calculate the number of branches that a file has. To fully measure McCabe, the all the possible branches should be calculated — if there are two case statements with the same variables, that should not increase the number of branch points in a system (although it can be argued that it will be more difficult to maintain, since if one case has to be modified, the other one might have to be modified as well).

Number of Parameters

We count the number of required parameters and parameters with defaults of each file, and add these together.

Number of exec resources

The number of exec resources is counted per file, and this metric rates how many files have (too) many execs. The derived threshold (what is too much?) is given in Chapter 7.

Number of filtered lint warnings

The number of filtered lint warnings (very common and 'harmless' errors that are probably turned off are left out) is used as a quality metric. We count the number of filtered lint warnings per file.

Fan-in

We calculate the fan-in of files. Files that are called or referred to very often may have consequences when changed for all of their callees, and the number of affected files scales with the fan-in.

Module Degree

The module degree serves to capture the architecture complexity that can be seen from the dependency graph. We calculate the amount of code inside a single module, and the degree (both fan-in and fan-out, but not self-references) of said module. A configuration where this metric scores low can show some architectural debt, i.e. an example is that most activities take place inside a single module. This metric is affected by system size – having a small system and a single module that is very tightly coupled is rated lower than a large configuration with a lot of coupling between modules. Dependency graphs are further discussed in section 6.3.

Configuration-level metrics

We use the Volume and Duplication metrics explained in Chapter 4.

- **Duplication** Duplication is measured by detecting duplicated blocks of at least 6 lines (excluding curly braces), and measuring for each line whether it exists inside a duplicated block. Only redundant lines are reported (we traverse files in order, so if a piece of code is flagged as originated in file A and found in file B, only the duplicate of file B is counted) We divide the number of redundant lines by total lines of code (we do not count lines with only curly braces in total lines of code). The percentage of redundant lines is reported for the entire configuration.

- **Volume** Volume is measured by adding up all the values calculated by the filelength metric.

5.5 Answer to Research Question 2

The research question we answer is *How can these quality measurements be combined into a measuring instrument?* By cloning all public Puppet repositories from GitHub and filtering them, and by applying the measurements on both the raw and the filtered dataset, we have measured the suitability of measurements from Chapter 4. This resulted in our quality model (table 5.5).

Measurement	**Implementation**
Filelength	The number of lines per file.
Complexity	The number of control statements per file, plus number of alternatives in case statements.
Number of parameters	The number of parameters per file.
Number of exec resources	The number of exec resources per file.
Number of filtered lint warnings	The number of filtered lint warnings per file.
File fan-in	The number of incoming references per file.
Module Degree	The number of incoming and outgoing number of references per module.
Duplication	The number of redundant lines of a code base divided by the number of lines of a code base.
Volume	The number of lines of a code base.

Table 5.5: Our quality model measurements and their implementation.

Chapter 6

Tool Implementation

In this Chapter, we show the design behind our tooling to measure the quality of Puppet configurations.

We implemented the measurements by extending Puppet-lint to generate more measurements, and wrote a custom Haskell program to aggregate the measurements and to draw dependency/call graphs. A high-level overview is provided in figure 6.1.

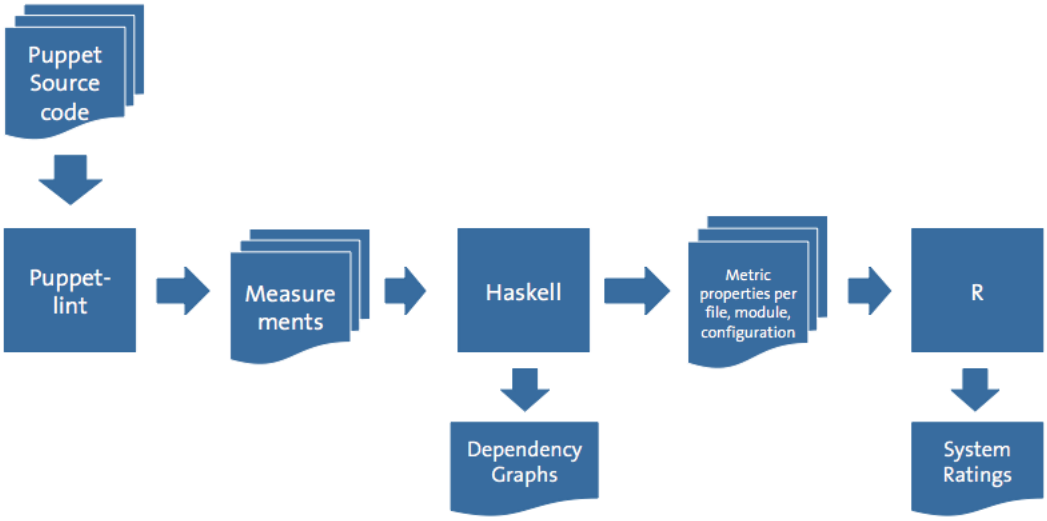


Figure 6.1: An overview of our tooling set-up. The input is Puppet source code, the output consists of dependency graphs and star ratings.

6.1 Puppet-lint

We opted to use Puppet-lint instead of writing our own tool, because it allowed us to reuse the parser. In addition to this, the program structure of Puppet-lint was easy to extend to simply add more measurements. This reduces the number of tools needed for the job, since the measurements and lint errors were done using the same tool.

We adapted Puppet-lint to automatically traverse an entire folder structure and check every

.pp file, but we excluded all filepaths that contained folders called ‘spec’, ‘tests’, ‘files’ and ‘examples’, since this is either test code or code that skews the metrics. For example, resources declared outside of classes happens often in files in the ‘examples’ directory, but since this is not used in production code we left it out of our measurements. Puppet-lint is not able to detect duplicated blocks, so we added a duplicated block detector as well.

6.2 Haskell

We parsed the measurements from our adapted Puppet-lint tool, and grouped the measurements by file. Then, making use of the filepath we analyze what module the file belongs to, and we group the the measurements of files that belong to the same module. Finally, we grouped the measurements of all modules that belong to the same configuration. Configurations without modules, i.e. a single module, are grouped together as a single module. We treated the main manifest as a module.

This allows us to automatically analyze very large numbers of repositories.

6.3 Graphs

Since a Puppet architecture is mostly defined by its modules and their relations, we tried to graph these relations to visualize the architecture.

6.3.1 Visualization

In order to rate the complexity of an architecture, and since we had everything in place to create the dependency graph, we decided to draw dependency graphs. This allowed for creating a metric to rate the complexity of an architecture.

We created module-level dependency graphs, since this allows for a high-level overview. The dependency graphs also allow for developers to inspect their architecture - what relations between modules are there that should not be there, or which modules will have to be edited if another module is changed or removed.

6.3.2 Constant Propagation

Since some Puppet setups had constructions like “ exec { $var : ... “, we implemented constant propagation to resolve these calls.

Since Puppet is declarative and pure, variables can only be assigned a value once. To propagate constants, we took all the variables and parameters of a file, took all the calls to externals (classes etc.), and propagated the variables that are used in the call. In the next iteration, the same thing happened, but the possible values of parameters are extended with all received constants. This process continued until a fix-point was reached. Whilst theoretically this slows down the implementation, in practice the time taken to run the constant propagation analysis was not noticeable. After this point is reached, we add the additional information gained to the resources and then create the associated file graph.

6.3.3 Calculating Dependencies

We resolve ambiguous calls by calculating the distance between the filepaths and assigning the call to the closest possible file. A distance of 0 means the same file, a distance of 1 means the same directory, etc. This was done to avoid having calls from third-party code to your own code.

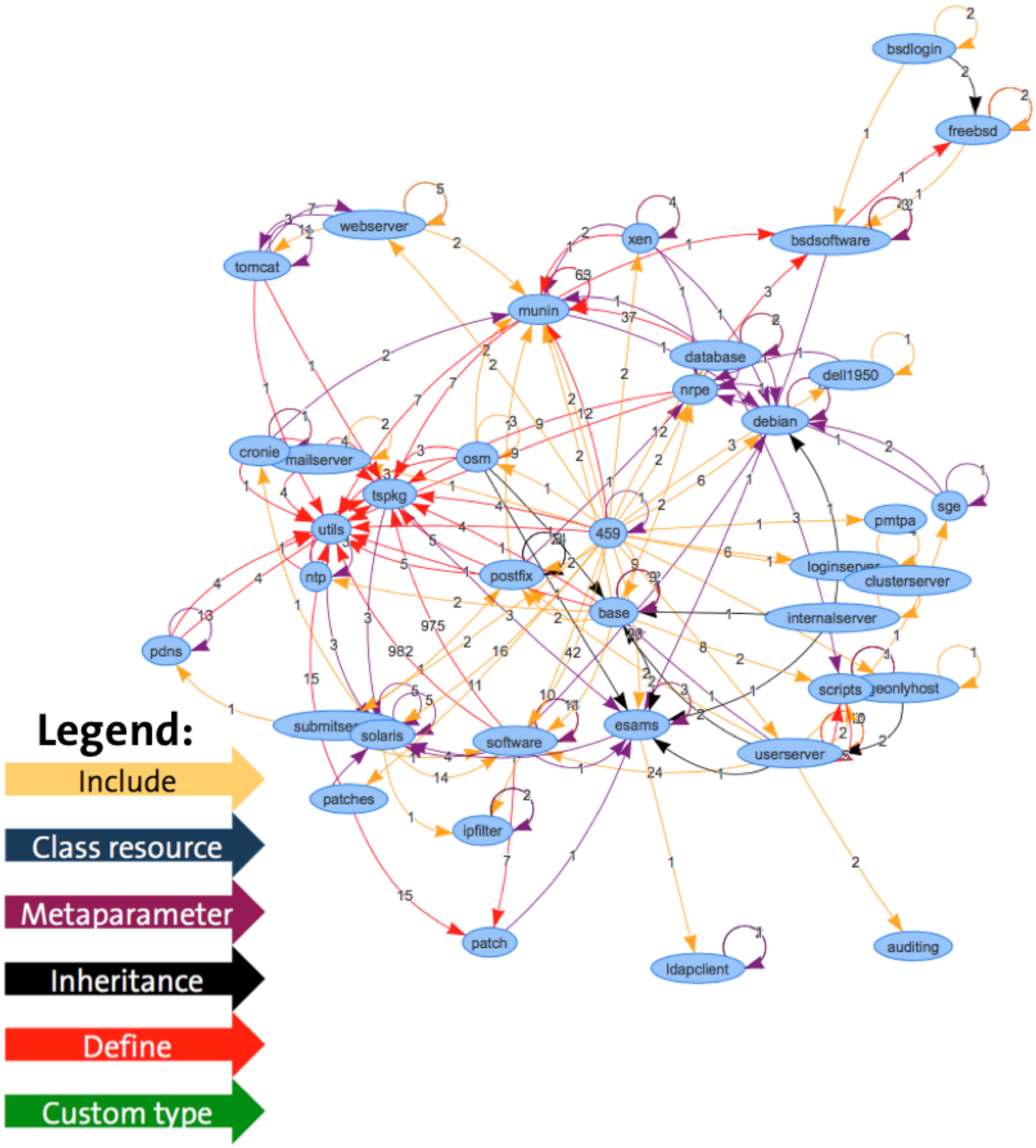


Figure 6.2: Dependency graph of system 429.

In the graph in figure 6.2, every node is a module. The node marked ‘459’ in the center is the main manifest. In this graph, there are no custom types or classes are defined as resource.

The advantage of having such a graph available is that it becomes possible to visualize code-level dependencies between modules, and this will allow you to see what changes might occur when a module is changed or removed and what will be affected. As far as we know we are the first to visualize Puppet module dependencies in this way.

Further extensions are to include the ‘contain’ statement, and also include arrows. Arrows in Puppet (not to be confused with arrows in the graph) are used to specify an ordering between resources (you cannot start a service if the package is not installed), however visualizing this is a challenge — an example of such a challenge: it is possible using these arrows for a resource which should be applied before any package resource. This would then mean that arrows would have to be drawn between this module and all other modules which have packages. This can make things difficult to see and interpret.

Other ordering mechanisms could be interesting as well, since they are now drawn as the same arrow (metaparameter).

Finally, a combination of data from Hiera and Metadata could be added as well – only Puppet-code level dependencies are drawn.

Chapter 7

Calibration of the measurement model against a benchmark set

In Chapter 4 we have laid out the metrics of the quality model. To see how the metrics perform on actual datasets, we have filtered the GitHub dataset to only show real-world Puppet configurations, since we want to create a model that works on this kind of code bases. We consider a 'real-world' Puppet configuration to be a configuration that is similar to what might be used at a company, and where daily maintenance can be expected. To select these repositories, we focus on the size — we expect that companies that manage their infrastructure with Puppet have large code bases. In this Chapter, we explain the filtering that we have done to arrive at this dataset and how we have calibrated the thresholds of the quality model against this dataset.

7.1 Benchmark Dataset

The configurations we selected were taken from the original dataset as explained in section 5.1. We took repositories that had at least 5000 lines of Puppet code (this excludes commentary, whitespace and lines containing only curly braces), and had a 'modules' directory. In a Puppet configuration, the default location of the modules is in the directory called 'modules'. By requiring this directory, we focus only on configurations and discard very large modules.

This resulted in 299 repositories. Since we cloned every GitHub repository, we also cloned a lot of forks (or identical repositories not marked as such). We pruned the dataset using total lines of code, the total lines and the percentage duplicated. Repositories with identical or very similar numbers were considered to be the same (or largely the same besides a main file that has a few configuration parameters edited).

We chose the number 5000 for multiple reasons. First of all, the number of repositories with less than 5000 lines of code was far greater than the number of repositories with 5000 lines of code or more. If smaller repositories were allowed, multiple things would happen. It is possible for a single, large module, to dominate the size of a repository. Then, the properties of that module will dominate the rating. Since we know that in general purpose programming languages, smaller systems are easier to maintain than large systems [23], this can skew the ratings of the benchmark. Since we want to review every repository of our benchmark manually to see if it is not a duplicate of another repository, we have to limit the number of repositories. Finally, since the participants in the validation study cannot review every repository, we have to make a selection. There is a possibility that the number of small repositories will be far larger

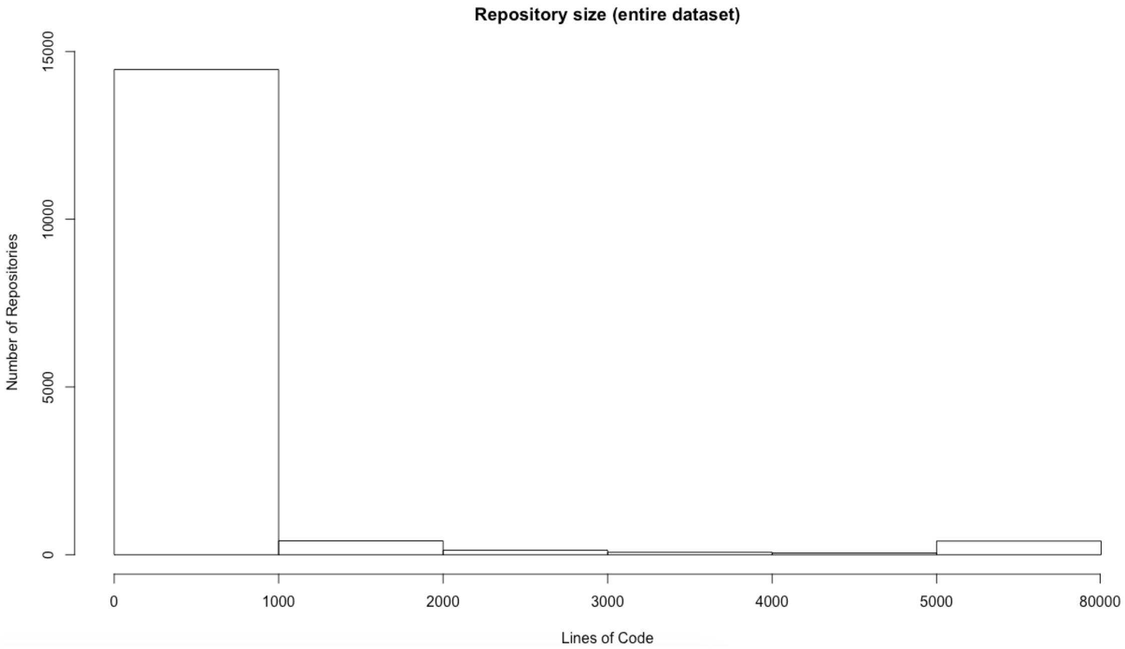


Figure 7.1: Lines of Code per configuration of the entire dataset. Note that the last bin contains every repository between 5000 and 80000 lines of code.

than the number of large repositories. If this model is to be applied in the real world, i.e. help a Puppet programming team get their code under control, we expect that this code base will be large. Thus it is more useful to train the model on larger repositories than very small ones. Whilst it is possible that a few repositories smaller than 5000 lines of code would still fit the criteria of being a 'real world' system, we expect this number to be low.

After pruning duplicated repositories, about 100 repositories remained. A lot of repositories were 'puppi' repositories which consisted of mostly third-party code, but slightly differed from each other so that they managed to get past the duplication checking earlier. We removed all 'puppi' repositories from our dataset. The downside of not removing repositories containing third party code is that the rating turns into a score of how much of your codebase is third party code, which should not be maintained by you (although third party code might have other problems) and it would not be useful to have this reflected in a quality rating. We show that third party code has different properties than self-maintained code in the next section. Puppi on itself is simply a module, and the stack (configuration) is mostly third party code, so removing this from our dataset does not mean that we have removed a useful configuration.

Since we cloned all repositories with git submodules, we removed all submodules and removed all repositories with 5000 lines of code or less.

We observed that in some repositories third party code was present alongside self-written code. We removed all modules that contained a 'LICENSE', 'CHANGELOG', 'CONTRIBUTING', 'CHANGELOG.md', 'CONTRIBUTING.md' file, and again looked at what contained more than 5000 lines of code. This resulted in 26 repositories. We manually reviewed each repository, and removed a few duplicate repositories. Repositories that were not configurations (simply large collections of scripts, not even a collection of modules) were also removed.

This resulted in 17 repositories. Our filtering was done in a rather iterative way. Initially

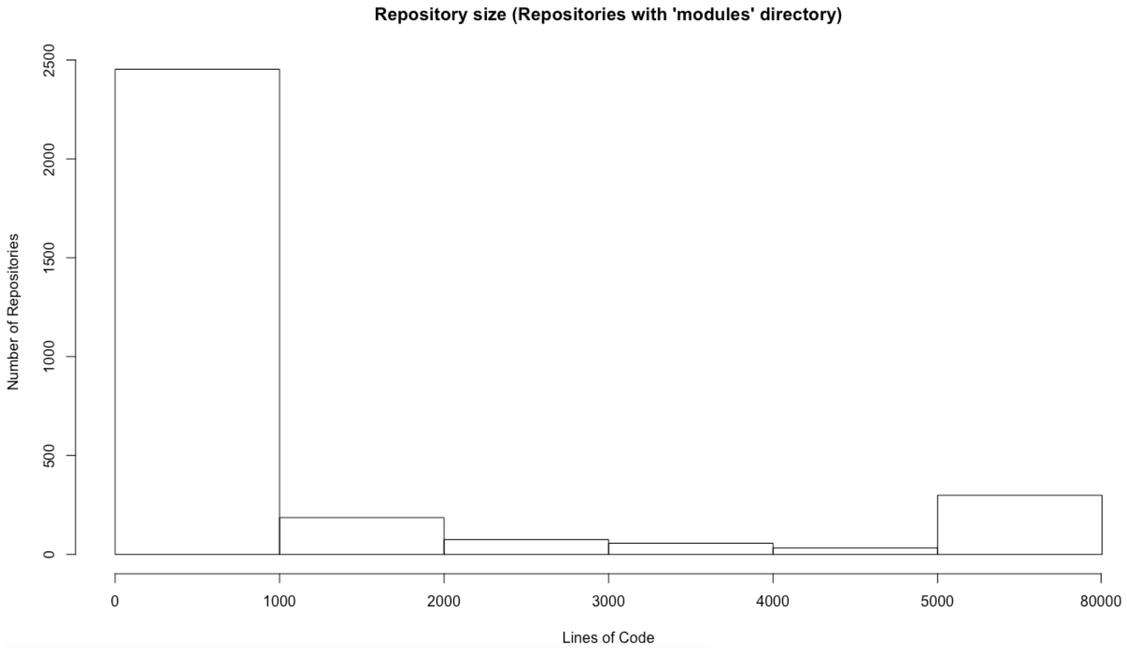


Figure 7.2: Lines of Code per configuration of every repository with a 'modules' directory. Note that the last bin contains every repository between 5000 and 80000 lines of code.

we applied very basic filtering (only have a modules folder so that this is a configuration), but after certain metrics started to show weird values (a large number of repositories had exactly the same scores) we kept filtering until the dataset did not contain any problematic artifacts and was useful to use as a benchmark.

In our selection, only a fraction of the original dataset of 15.540 repositories was selected. The reason for this, besides that we only selected repositories that have a 'modules' directory, and that there was a large amount of duplication in the 299 repositories with a 'modules' directory, is that the majority of Puppet repositories have 1000 lines of code or less.

It is infeasible to inspect every repository to see what it does, but manual inspection of a small subset of the repositories with less than 1000 lines of code, shows that those small repositories are used to accomplish a very small task. Examples are configurations that only manage 5 modules, or repositories that only contain a template.

7.2 The difference between third-party code and own code

One of our observations is that code from third-party sources (modules specifically) is vastly different from the code that has been written for own use.

As mentioned in the survey, third-party modules try to handle every use case, support more Operating Systems and have more parameters / options. For all assumed non-third party modules we have drawn quantiles of filesize, complexity and parameters, and we have done the same for all Puppet approved and supported modules that contain Puppet code. These quantiles are shown in table 7.1, table 7.2 and table 7.3.

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Own	0.0	0.0	0.0	2.0	4.0	7.0	12.0	21.0	48.0	2416.0
third party	3.3	10.6	22.0	30.6	54.0	87.6	112.2	170.4	331.5	633.0

Table 7.1: Quantiles of parameters per module

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Own	10	18	29	44	63	92	141	212	406	19262
third party	37	119	209	276	479	596	826	1261	1935	3869

Table 7.2: Quantiles of lines of code per module

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Own	1	1	2	4	5	7	12	20	39	1301
third party	6	14	22	30	47	73	111	140	174	480

Table 7.3: Quantiles of complexity (Total number of branches through code) per module

As is visible, most non-third party modules are smaller, have fewer parameters and are less complex than third party modules. On the other hand, there are non-third party modules that are much larger, have more parameters and are more complex than third party modules.

7.3 Rating repositories

To use the metrics, we made use of risk profiles [20]. With risk profiles, it is possible to rate how much of a certain code base falls into either the low risk, medium risk, high risk or very high risk category, and using these percentages for all systems, it is possible to make a comparison and give a quality rating. The advantage of risk profiles is that instead of raising an error saying "this file is too long" after it passes a certain threshold, which might appear hundreds of times when first analyzing a codebase, you can accurately see how severe the problem is — are the files just a bit too long, or are the files extremely large?

We derived Puppet-specific thresholds for the metrics in our model, in an approach similar to Alves et al. [9] and applied these thresholds. In our case, we opted for thresholds that were very different (widely spread) from each other. Where possible, we made sure that the threshold for medium/high risk was twice as large as low/medium risk, and high/very high risk was as least trice as large as low/medium risk.

We then created a risk profile per metric by deriving the 70/80/90% values of a configuration. Then, using this risk profile, we derive the thresholds to make the data fit a 5/30/30/30/5% distribution. In this case, we consider the 5% best systems to have 5 stars, the next 30% of systems to have 4 stars, etc., and finally have the 5% worst systems to have 1 star. After fitting the systems to the 5/30/30/30/5% distribution, we derive the thresholds to get a certain star rating. Thus, if have exactly the same or less than the 5-star threshold, you will get 5 stars. If you have less than the 4-star threshold, but more than the 5-star threshold, your rating will be between 4 and 5. If you have more than the 2-star threshold allows, you will score 1 star.

Consider for example complexity. The risk profile we derived is shown in table 7.4, and the thresholds for the star-ratings are shown in table 7.5.

Risk Category	Complexity Value
Low Risk	1-7
Medium Risk	8-20
High Risk	21 - 34
Very High Risk	34+

Table 7.4: Risk profile for complexity

Stars	Medium Risk	High Risk	Very High Risk
*****	2.4	0.0	0.0
****	13.0	1.5	0.0
***	69.0	53.2	7.4
**	86.5	76.2	34.0

Table 7.5: % of LOC allowed per risk category for Complexity

In table 7.5, if you have at most 2.4% of your lines of code in files of a 'medium risk' complexity, and no LOC within files of 'high risk' and 'very high risk', you belong to the best 5% of the observed systems. To belong to the worst category (1 star), you have to score below the thresholds of the 2 star cutoff. There is no limit on the amount of 'Low Risk' code that can be present in a configuration. Another example is filelength. Whilst we used a 70/80/90% values, as proposed by Alves et al., for our other metrics, for filelength we used 25/50/75% instead, since the thresholds from 70/80/90% were too close together.

The thresholds for filelength are shown in table 7.6, and the thresholds for the star-ratings are shown in table 7.7.

Risk Category	Complexity Value
Low Risk	0 - 45
Medium Risk	46 - 110
High Risk	111 - 345
Very High Risk	345+

Table 7.6: Risk profile for filelength

Stars	Medium Risk	High Risk	Very High Risk
*****	31	3.3	0.0
****	56	26.0	7.3
***	90	77.5	14.0
**	95	89.5	45.0

Table 7.7: % of LOC allowed per risk category for Filelength

For execs, the thresholds for the risk categories are shown in table 7.8, and the thresholds for the star-ratings are shown in table 7.9.

Risk Category	Complexity Value
Low Risk	0
Medium Risk	1
High Risk	2
Very High Risk	3+

Table 7.8: Risk profile for execs

Stars	Medium Risk	High Risk	Very High Risk
*****	0.50	0.00	0.0
****	5.90	2.50	0.7
***	54.95	51.25	1.8
**	77.45	75.25	7.6

Table 7.9: % of files allowed per risk category for the number of execs

For execs, we look at the number of execs per file (they are not weighted by LOC). Whilst the number of execs allowed per file might seem very low, the thresholds for the star rating shows that you are still allowed to use them sparingly without getting punished harshly.

We rated the number of parameters, the number of filtered lint warnings, fan-in and module degree in the above manner. For execs, we did not aggregate by LOC but simply looked at the file. We rated volume by putting the systems in 5/30/30/30/5% bins, ordered by volume, and gave a rating based on that. Duplication was rated in a similar manner. For our experiment, we used the average of the ratings as a quality rating. The thresholds derived here were only for 17 systems. It is very likely that when more systems are taken into account, these values will change.

Chapter 8

Quality Model Validation

In this Chapter, we will describe how we validated our quality model, and answer research question 4. For our validation, we had the option of doing either a survey or hold interviews. We decided upon the latter, for multiple reasons. First of all, time was an issue — preparing the survey with proper questions is non-trivial (we wanted to have experts rate or compare code) which would be very time consuming. Second, we could not think of a use of having respondents in a survey rate code beyond single files, but our model is made to rate entire configurations and it might be very difficult to translate results from the survey into whether the quality model is good or not. Third, we could not see what respondents did, explain a question further if they did not understand it, or ask additional questions based on the answer of the respondent. When holding interviews, we would not have those problems. However the disadvantage is that the population of the interview is smaller than the population of a survey.

In this Chapter, we explain the interview, discuss the results of the code comparison and summarize the responses to the setup of the questions. The full responses together with comments made during the comparison can be found in Appendix B.

8.1 Validation Setup

The quality model was validated with semi-structured interviews, with different groups of people. We interviewed several SIG employees with Puppet knowledge, as well as a Puppet programmer from a different company, an external researcher with knowledge of Puppet and two employees of Puppet (formerly known as Puppetlabs).

The goal of the validation is to both see if the proposed quality metrics are good enough predictors of code quality, but also to see if there are potential flaws in the model. For example, to discover whether trying to improve on certain metrics will not affect or negatively affect the maintainability of a system, or if certain metrics are simply not suited to the language or the domain of configuration management. It is also be possible that some metrics should be weighted more heavily (i.e. volume or a complex architecture) than is currently implemented.

The interview consists of both questions and a comparison between systems. We rated 17 systems using the metrics described above, and used the average to rate them. There was no system that scored excellently in all the metrics, nor was there a system that scored terribly in all of them. We thus split the 17 systems into groups of 20% of every system, by quality. The 20% worst systems would be in the bin ‘worst systems’, etc. We then selected from every bin the most up-to-date system. Selecting the most up-to-date system from either the date of the validation studies (early may) date or the date of the snapshot (1 Feb) resulted in the same systems being selected.

We used 5 systems, because there is a limited amount of time available for every interviewee to rank the systems. The time scheduled for the interview was one hour. With 5 systems, we used on average 45 minutes for the comparison and 15 minutes for the questions.

We compared the systems by ranking instead of rating. The experts that we are interviewing are experts on Puppet, and whilst some have experience with using automated tooling to rate software systems, they form a minority. We expect that if a rating is used, there is a risk of ratings being all over the place, which would make it very difficult to analyze. We have opted to use a ranking instead, similar to Cox et al. [24].

In advance, we sent the participants the GitHub links of the snapshots of the repositories that we used [21], and asked them to take a look at them, and if possible rank them. Everyone looked at them before the interview, ranging from 5 minutes to two hours. At the start of the interview, some had already made up their mind on the ranking whilst others had not yet decided on a ranking.

During the interview, the following questions were asked:

1. What maintainability problems have you encountered in Puppet?
2. A lot respondents to the survey have said that overuse of execs is a bad practice. Do you agree? If so, what issues arise when execs are overused?
3. What do you think is missing from the model? What should be changed?
4. What do you think should be added to the model?
5. Do you think the model is useful in practice?

At the start of the interview we first explained what maintainability is according to the ISO 25010 standard – making changes, upgrades, updates, bug fixes, etc. Different people have different interpretations of quality, so the focus of the interview was on maintainability. After this, we asked questions 1 and 2.

Then, we asked the interviewees to make a ranking of the five systems, with the best system being first and the worst system last, based on their maintainability and quality. We provided descriptive statistics of the related systems and a call graph for every system.

After this ranking, we showed them our model and that we use risk profiles, and asked questions 3, 4 and 5. This concluded the interview.

8.2 Validation Results

The summarries of the interview and the remarks made by interviewees on repositories are included in appendix B.

8.2.1 Ranking

System	GovUK	RING	Fuel_infra	Wikimedia	Kilo-Puppet
Rating	3,57	3,47	3,05	2,91	2,34
Rank	1	2	3	4	5
Interview 1	3	1	2	4	4
Interview 2	1	2	3	4	5
Interview 3	2	1	3	5	4
Interview 4	1	4	3	2	5
Interview 5	1	2	4	3	5
Interview 6	1	2	3	4	5
Interview 7	1	2	4	2	5
Interview 8	2	1	3	5	4
Interview 9	4	1	2	5	3
Median Interview Rank	1	2	3	4	5

Table 8.1: This table shows the name of the systems, the rating according to our model, the associated rank of the system, the rankings the interviewees gave the systems and the median of the rank of the interviewees.

As is visible, there are differences between the raters in rankings of the systems. These differences can be attributed to a few different factors. Some interviewees put a very large emphasis on the volume of the codebase, and mostly used that to rank the codebases. Some interviewees considered the presence of tests and documentation to be very important, while others did not. In addition to this, the ratings of some systems were very close to each other, such as with RING and GovUK. Some interviewees ranked some repositories as equally maintainable. Finally, there might have been differences in the definition of maintainability used. Some interviewees considered "how easy would this be to start maintaining", and others considered "how easy is it to maintain this system".

The median of the interview ranks is equal to the rank given by the tool, which shows that our ranking fits the interview ranking. We have used the median since the rank is an ordinal variable, and if the mean of this data is taken, this might be heavily influenced by outliers (For example, an interviewee might rank a system as '5', but can still consider it to be of high quality). It would have been possible to take the mean of this data if a *rating* was asked, since in that case differences between systems can be expressed more clearly by interviewees instead of giving a ranking. In this case, if there are outliers (i.e. giving a system considired to be high quality by other interviewees a '1.0' rating), those outliers are not the cause of the method used (as might be the case with ranking) and should be taken into account.

The inter-rater agreement using Kendall's W (we did not include our tool as a rater) was 0,589, with a p-value of $2 * 10^-4$. This means that between the interviewees there was strong consistency [16]. This is reflected in the data and we have given some explanation for this disagreement earlier.

8.2.2 Answers to questions

We summarize the answers given to the questions asked during the interview. The interview transcripts can be found in Appendix B.

What maintainability problems have you encountered in Puppet?

Different types of problems were reported. Upgrading third party libraries was reported, as well as lacking tests or keeping up with tests.

A lot respondents to the survey have said that overuse of execs is a bad practice. Do you agree? If so, what issues arise when execs are overused?

The overuse of execs is considered bad all across the board. Main reasons are non-idempotence, outside of the declarative model and it is not well supported in the IDE or the language. That being said, most people indicated that exec's aren't that bad – it should be used with care. This is in agreement with the survey answers.

What do you think is missing from the model? What should be changed?

- Execs: Instead of a rating, a violations category for this should be considered. Additionally, look for execs that always run, either by analyzing the resource itself or the output of a Puppet run.
- Lint Warnings: Some warnings might be disabled, which can differ from project to project. In addition, some warnings are unavoidable or false positives. An example is the quoted boolean warning. Another example is inheriting from a params class — doing this makes your code incompatible with Puppet 2.7, which is probably not going to be an issue if you already past 2.7. Another example is having a quoted boolean in the code. Puppet-lint will raise a warning if this is found, but it does not check how this value is used — if a string hash has to be specified to be used in a template, this warning is a false positive.
- Parameters: Having a high number of parameters is considered less important — some indicated that the number of parameters should have different thresholds, or that the number of required parameters should be taken into account – i.e. is it possible to use the module by just including it.
- Duplication: Duplication is a useful metric – but either the block size should be changed (things may be considered duplication that aren't, for instance resources with long parameters), or a better clone detector should be implemented so that cloning and reordering parameters (syntactically different, semantically identical) can still be detected, since those clones still suffer the same disadvantages as regular duplication but are not spotted using the current measurement.

What do you think should be added to the model?

- Metric to rate a module: In some cases a 'god module' was present in the configurations. Having a module with too many responsibilities is not following the best practice of modules only having one task, and can become a maintenance risk. Whilst module degree measures whether a module has too many connections, if the module has a few connections but is very large, this is also an indication of too many responsibilities.
- Testing: Currently tests are not taken into account, but from the answers we got – both with maintainability problems and this questions – a metric to rate how well testing is done can be added. Different metrics are suggested, such as test code volume, assert density, but also to look for 'node coverage'. It could be argued that code that is the basis of the infrastructure and present on many nodes should have higher test coverage (or be more maintainable in general) than code that is only included a few times.

In the SIG model, test code is not included (whether the code does as required is a part of Functional Suitability in the ISO standard, not maintainability) although Building Maintainable Software [23] recommends a coverage of 80%.

- Library Management: Are third party modules that are included in code (if any) marked as such and never edited? The best practice is to never edit third-party libraries. Downsides of editing third party modules is that it becomes difficult if not impossible to combine it with upstream changes, it adds to the maintenance burden (since they are typically larger and more complex than what you would write yourself), and if it is not possible to combine it with upstream changes, you will not get security patches or feature updates. In addition to this, new employees will also have to get used to your edited third party libraries instead of the ones that are commonly used (and the new employee is most likely familiar with).
- Documentation: From the comparison and the answers to these questions, many programmers found documentation to be a very important aspect. Particularly if a codebase is very large, having documentation to explain where to start and how code is developed is very important. An additional challenge is making sure that what is written in the code and in the documentation remain consistent.

Other quality related aspects should be considered as well when judging a configuration, such as how passwords and other secrets are handled, end-of-life third-party modules, and if there are certain hardcodes. The last part is particularly difficult since different people use different ways, and having regular expressions search for emails, URLs and IP-addresses might return a lot of false positives – either because they are 127.0.0.10 or they are in accordance with the policy that is used for handling data.

Do you think the model is useful in practice?
Yes, every respondent considered the metrics to be useful. The metrics are considered useful to see where the problems in a codebase are, and is considered useful as a general indication of quality. Improving a metric blindly is considered a bad idea – it might not be worth the effort, or might not make the code better — it is a judgment call. One interviewee indicated that he would want to use a tool like this for his own organization.

To add to the above question, at this point in time the tooling does not yet show where the main problems in your codebase are. To find files that have a large complexity value or a large number of redundant lines, it is possible to find that information from the CSV that is used to rate the codebase.

8.3 Answer to Research Question 3

The research question was: *Are the quality measurements from RQ2 a good indication of Puppet code quality?* The research question consisted of two subquestions:

RQ 3.1 Is the combination of measurements from RQ2 minimal and complete?

RQ 3.2 Is the combination of measurements from RQ2 considered useful in practice by Puppet experts?

Is the model minimal and complete? We do not consider our model to be minimal or complete. The case can be made for parameters to be left out – there is a high Spearman correlation between both scoring great on parameters and complexity, which is 0.8. Finally, parameters were mentioned to be not as important (see interview 3). Parameters are however a useful measure to compare between third party and own code, as we showed in section 7.2.

A violations category should be added in favor of some measurements such as lint-warnings/errors (and make lint optional) and execs. Finally a measurement to rate modules on their size/structure should also added, as well as a testing metric.

Are the measurements useful in practice? Every interviewee indicated that he considered the model useful in practice. The measurements are considered useful to compare repositories against each other, and to identify technical debt. Blindly following the metrics to get a better score was not considered a good thing. The use of risk profiles was received very well by those unfamiliar with it, and some experts liked that the number of metrics is small and easy to measure.

To answer research question 3, are the measures from RQ2 a good indication of Puppet code quality? Yes, we think they are. Our ranking shows that the prediction we made was as good as possible, but since the model is not considered minimal or complete, we think there is room for improvement in the model.

We think the model can be improved because the respondents mentioned that different metrics should be added / removed, and this would make the model more useful in practice since it can spot more technical debt.

At this time aspects are weighed uniformly, and a weight could be added — i.e. take file metrics together instead of letting them count separately, this weighting will have to be validated as well, which can be done as future work.

With the changes made it is possible that the scores of the repositories change, and that using the same selection criteria, different repositories would be selected that would lead to more agreement between raters.

Chapter 9

Related work

Sharma et al. [22] have developed the tool 'Puppeteer' to detect code smells in Puppet. They analyzed common smells in software engineering and Puppet-related smells and measured them on a number of GitHub repositories. They analyzed 4.621 repositories from GitHub, whilst our initial dataset was larger (15.540), but the number of repositories we used for our experiment were much smaller, being only 17.

Xu & Zhou [6] have researched errors in configuration management, so-called misconfigurations, and have analyzed their causes. Just as unmaintainable software can be a financial disaster, misconfigurations can also cost a lot of money. They analyze why things are misconfigured, and also suggest solutions to some of these problems. Their subject differs from ours, but we consider it a useful source to help troubleshoot difficult errors.

Alves et al. [9] have shown how to derive thresholds from metrics for the SIG quality model. They demonstrate how to automatically derive thresholds, and provide justification for why the thresholds in the SIG model are what they are. We followed a similar approach to pick the thresholds for our quality model.

Cox [24] has used expert opinion to rate a dependency freshness metric. The dependency freshness metric tries to capture how well developers keep third-party dependencies of their software system up-to-date. Their validation approach was similar to ours, using both semi-structured interviews and ordering given examples.

Lampasona et al. [3] have proposed a way to perform early validation of quality models, based on expert opinion. "We propose validating quality models with respect to their completeness (it addresses all relevant aspects) and to their minimality (a model only contains aspects relevant for its application context)." We used minimality and completeness in our validation interview, asking what could be removed/changed, or what should be added.

Building Maintainable Software [23] was a great resource for explaining the measurements in the SIG model. The book explains ten guidelines on how to build maintainable software, and whilst other papers on the SIG model explain the implementation, the book also provides justifications for why certain metrics are used and also demonstrates what happens when things go wrong.

Chapter 10

Conclusion

In this thesis, we have presented a quality model to assess the quality of a Puppet configuration. In this Chapter, we will provide a summary of the work we have done, reiterate the answers to our research questions and state our contributions. We also discuss other Puppet-related quality aspects that we did not measure, as well as discuss our identified threats to validity and future work.

10.1 Summary

We have analyzed the SIG quality model and held a survey to find Puppet-specific quality aspects, and we selected quality measurements from the SIG model as well as quality aspects from the survey.

We created tooling to measure these quality aspects, and we pruned the set of measurements of measurements were deemed not to contribute to the quality assessment. In addition to this, our tooling is able to generate dependency graphs based on relations between modules, and to the best of our knowledge, is currently the only tool that does this.

We mined GitHub for every public Puppet repository, and discovered that sometimes code marked as Puppet was not in fact, Puppet. We filtered the repositories, removing forks/duplicates and focused on configurations, that had enough self-written or self-maintained code. We ended up with 17 systems suitable for our empirical study.

From these 17 systems, we ran the measurements and made a selection of 5 systems, of different quality. We used these 5 systems in a validation session, which consisted of a structured interview. In addition to asking questions about maintainability and the model, we also asked experts to judge the 5 repositories based on their quality and maintainability. The ranking was done mostly on maintainability, but sometimes a quality issue was found that was considered severe enough by the interviewee to adjust the rank.

From the ranking, we can derive that our tooling manages to accurately predict the perceived quality as reported by the expert, although there was no total consensus between experts. In some cases the experts used other measurements to rank the codebases than we used in our tooling. Since our prediction is a match between the median of the predictions of the interviewees, whether the model has to be adjusted to take this into account is a subject of further research.

In addition to ranking the repositories, we also asked if measurements could be removed or should be added, and if the measurements would be useful in practice – both as a guide to improve code as well as to measure the quality of a repository. Some changes were suggested, both in removing and changing current metrics, and adding metrics to the model. The metrics were perceived as useful to use to improve code, identify technical debt, and easy to measure, although it was also mentioned that improving the code is a trade-off between gained maintainability and expended effort.

10.2 Answers to Research Questions

In this section, we repeat our research questions and their answers.

10.2.1 Research Question 1

Research question 1 is: *What quality aspects of Puppet code quality can be measured?* This question is split into two subquestions:

RQ 1.1 What quality aspects of Puppet code quality can be measured at the file level?

RQ 1.2 What quality aspects of Puppet code quality can be measured at the module / configuration level?

To answer this question, we looked at existing knowledge of quality, specifically the SIG model, and we held a survey among Puppet developers on GitHub. From this knowledge, we created a list of all possible measurements and selected our measurements by applying the following criteria:

- Easy to measure
- Only measures Puppet source code
- Metric is applicable on almost all codebases
- Metric is promising w.r.t. quality / maintainability
- There is a clear opinion on what is good quality for the metric

Examples of file-level metrics that we selected are filelength, complexity, and the number of exec resources. An example of a configuration-level metric is the volume of the configuration.

We have answered Research Question 1 in Chapter 4. Table 4.1 shows the quality aspects of Puppet code that we measure.

10.2.2 Research Question 2

Research question 2 is: *How can these quality measurements be combined into a measuring instrument?*

We took the set of metrics from research question 1 as a basis to create our model. By cloning all public Puppet repositories from GitHub and filtering them, and by applying the measurements on both the raw and the filtered dataset, we have assessed the suitability of measurements from Chapter 4. We pruned metrics that were deemed not suitable from our set of measurements from research question 1, formed our quality model by taking the metrics that we did consider useful. Our quality model is shown in table 10.1.

Measurement	**Implementation**
Filelength	The number of lines per file.
Complexity	The number of control statements per file, plus number of alternatives in case statements.
Number of parameters	The number of parameters per file.
Number of exec resources	The number of exec resources per file.
Number of filtered lint warnings	The number of filtered lint warnings per file.
File fan-in	The number of incoming references per file.
Module Degree	The number of incoming and outgoing number of references per module.
Duplication	The number of redundant lines of a code base divided by the number of lines of a code base.
Volume	The number of lines of a code base.

Table 10.1: Our quality model measurements and their implementation.

The filtering of our dataset and assessment of the suitability of the measurements from Chapter 4 is done in Chapter 5. How we implemented this model in tooling is explained in Chapter 6.

10.2.3 Research Question 3

Research question is: *Are the quality measurements from RQ2 a good indication of Puppet code quality?* This question is split into two subquestions.

RQ 3.1 Is the combination of measurements from RQ2 minimal and complete?

RQ 3.2 Is the combination of measurements from RQ2 considered useful in practice by Puppet experts?

This question is answered in Chapter 8. We have demonstrated that the quality measurements are a good indication of quality by interviewing Puppet experts. We asked 9 Puppet experts to rank 5 repositories provided to them, and we show that the median of their rank and the rank that our tooling provides are identical. Although there is no perfect consensus on the rankings of repositories between interviewees.

The model is not minimal or complete — interviewees suggested that some metrics could be removed, and in addition to this, some metrics could be added.

An example of a metric that could be removed is filtered lint warnings, since this is a tool-dependent metric in the model. Should developers switch to a different tool, the model would have to be adjusted as well. It also depends on how the tool is configured, since some options can be turned off. It can still be measured to see if programmers are using lint and where warnings are, but to have it in the model introduces some risks.

Examples of metrics that can be added are a metric to rate the modularity of a module, a metric to rate the quality of testing done or add documentation to the model (although automatically measuring that poses difficulties).

The experts considered the model useful to apply in practice, and experts considered the model useful as a general indication of Puppet code quality.

10.3 Contributions

To the best of our knowledge, there has been no prior research done on code quality in Puppet configurations. Our survey gives insight to what code quality means among Puppet developers. Our model and tool allow for automatically measuring and rating Puppet code quality. Whilst there are generic quality models available, our quality model is Puppet-specific and thus does not suffer from generic metrics that are difficult to translate (as discussed in section 1.3), and it also measures Puppet-specific quality issues rather than generic quality issues (an example is the exec resource). Our tooling also is able to create dependency graphs of Puppet setups on a module level, which can allow developers to gain insight into dependencies of their codebase.

10.4 Discussion

In this section, we discuss other quality aspects that we consider important in Puppet that we have not measured, and we discuss threats to validity that we identified.

10.4.1 Other Puppet quality aspects

There are some Puppet-related quality aspects that are not measured, but we still consider them important to keep in mind.

Automated Testing

Automated testing is probably the most important aspect, both on a module level (unit / acceptance tests) and on a larger configuration level, to see if the applied configuration is correct. Manually checking that changes or upgrades made have succeeded can be prohibitively time-consuming. In addition, to be sure that the configuration works as expected is very difficult without tests.

Security

How secrets such as passwords, certificates and keys are handled is an important quality aspect, and should be taken into account when looking at a configuration.

Continuous Deployment

Deploying manually to a large number of different servers, on a daily basis is simply impossible for a cost-effective price. The amount of manual effort to deploy a change should be as close to zero as is possible.

Library Management

Tracking what third party libraries you have, and having a plan to upgrade them ensures that you have the latest fixes and updates. If you edit those libraries, that goes away, and this adds to the maintenance burden, or can cause issues when upgrading.

Up-to-date Puppet

Make sure that an up-to-date version of Puppet is used. This makes it easier for new people (that are probably not used to older versions of Puppet) to join, as well as have the newest features and security updates available. If you are using an older version of Puppet, it can be challenging to upgrade it to the newest version.

10.4.2 Threats to Validity

We have identified two threats to validity. The first one is due to the repositories that we used, since they are small in number. The other threat is that whilst we have shown that our ranking fits the ranking of the interviewees best, we lack proof that an increase in the metrics translates to an increase of the quality and maintainability of the codebase.

Repositories used

The number of configurations that were usable for a benchmark was low. This might mean that the benchmark has a certain bias towards what happens in those configurations, and could mean that it does not translate well into judging quality of non-public configurations.

In addition, some repositories lacked documentation. This was taken into account by many of the validators as a big minus, which may affect the validity of the score. Having documentation makes it easier to understand what is going on in the code, and is considered a big plus to be able to start maintaining it and getting familiar with an unknown code base quickly.

According to the aggregated metrics, none of the repositories were either very good or very bad. We split the repositories in parts of 20% and then picked the most up-to-date repository of each. This might also mean that some repositories were close in quality, but because of the number of repositories this might mean that the selected repositories were not very different according to the model.

Using more repositories, and aggregating the metrics in a different manner – for example using weights, or aggregating the metrics before averaging them, can address the above issues.

Lack of real-world proof

It is not demonstrated yet that configurations that score high or low are indeed more or less maintainable.

To address this, an option is to track the time it takes to make changes and fixes from repositories and see if there is a noticeable difference – are high-quality repositories easier in maintenance than low-quality repositories? This is similar to the research of Bijlsma et al. [1] for the SIG quality model.

Another option is to monitor teams over time with tooling and perform a study to see if the tooling helps teams in writing better Puppet code.

This will address the lack of real-world proof, and can show that our model works to make code more maintainable.

10.5 Future Work

The current model can be improved by doing the following two things:

10.5.1 Change model

The model should be adapted with the changes proposed in Chapter 8. This means switching to violations for some metrics, and adding metrics that address the presence of a god module. A well-thought out weighting should be added as well.

10.5.2 Expand model with testing

The model should take testing into account, by adding a testing metric. This can either be test code volume compared to lines of code, to consider the presence of tests, or something more complicated. A case can be made that code that is included on a very large number of nodes (in the case of a large setup) should have a high test coverage.

Bibliography

[1] D. Bijlsma, M. Ferreira, B. Luijten, J. Visser. ”Faster issue resolution with higher technical quality of software”, Software Quality Journal, June 2012, Volume 20, Issue 2, pp 265-285.

[2] http://www.slideshare.net/PuppetLabs/modern-module-development-ken-barber-2012-edinburgh-Puppet-camp

[3] C. Lampasona, M. Kläs, A. Mayr, A. Göb, and M. Saft. ”Early Validation of Software Quality Models with respect to Minimality and Completeness: An Empirical Analysis.” Software Metrik Kongress. 2013.

[4] https://docs.Puppet.com/Puppet/latest/reference/lang_classes.html

[5] https://github.com/Puppetlabs/Puppetlabs-firewall/blob/master/manifests/linux/gentoo.pp

[6] T. Xu and Y. Zhou. ”Systems Approaches to Tackling Configuration Errors: A Survey .” ACM Computing Surveys 47, no. 4 (July 2015).

[7] https://github.com/Kmann180/WormsRemake/blob/master/WormsRemake/packages/MonoGame.3.2.2/content/net40/Game1.cs.pp=

[8] https://docs.puppet.com/facter/

[9] T. Alves, C. Ypma, and J. Visser. ”Deriving Metric Thresholds from Benchmark Data”, In proceedings of the 26th IEEE International Conference on Software Maintenance, 2010.

[10] http://puppet-lint.com

[11] https://www.Puppet.com

[12] R. Krempaska, A. Bertrand, C. Higgs, R. Kapeller, H. Lutz, and M. Provenzano. ”How to maintain hundreds of computers offering different functionalities with only two system administrators.” ICALEPCS. 2011.

[13] J. Visser. ”SIG/TÜViT Evaluation Criteria Trusted Product Maintainability.” Software Improvement Group, 2015.

[14] https://docs.Puppet.com/Puppet/latest/reference/lang_resources.html

[15] https://docs.Puppet.com/guides/style_guide.html

[16] C. Griessenauer, J. Miller, B. Agee, W. Fisher, J. Curé, P. Chapman, P. Foreman, W. Fisher, A. Witcher, and B. Walters. ”Observer reliability of arteriovenous malformations grading scales using current imaging modalities”. Journal of neurosurgery, 120.5, 2014.

[17] G. Gousios, M. Storey, and A. Bacchelli. ”Work Practices and Challenges in Pull-Based Development: The Contributor’s Perspective”, in Proceedings of the 38th International Conference on Software Engineering, 2016.

[18] http://evanderbent.github.io/Puppet/2016/01/28/Puppet-code-quality-survey.html

[19] https://gi thub.com/evanderbent/PuppetSurveyData

[20] T. Alves, J. Correia, and J. Visser. ”Benchmark-based Aggregation of Metrics to Ratings”, In Proceedings of the Joint Conference of the 21th International Workshop on Software Measurement (IWSM) and the 6th International Conference on Software Process and Product Measurement (Mensura), pp20-29, IEEE Computer Society, 2011.

[21] https://github.com/fuel-infra/Puppet-manifests/tree/27b65fa9767ec6e4f373d25779c14e04622a06cb
https://github.com/alphagov/govuk-Puppet/tree/03c24266624f001add066fabd5de0bbf583e68c7
https://github.com/wikimedia/operations-Puppet/tree/f0b468a92663f0306fb4039cd3ab87b6677b48cc
https://github.com/CCI-MOC/kilo-Puppet/tree/ccb5ac5f5d02a2b3f71d7ef7eabe2be00f24fe66
https://github.com/NLNOG/ring-Puppet/tree/d1a9a3a579788213205efa2a38d80f30b031c6e0

[22] T. Sharma, M. Fragkoulis, and D. Spinellis. ”Does your configuration code smell?”, in 13th international conference on Mining Software Repositories (MSR), 2016.

[23] J. Visser, P. van Eck, R. van der Leek, S. Rigal, and G. Wijnholds. ”Building Maintainable Software – Ten Guidelines for Future-Proof Code”, O’Reilly Media, 2016.

[24] J. Cox. ”Measuring dependency freshness in software systems”, Master’s thesis, Radboud University Nijmegen, 2014.

[25] https://www.ansible.com

[26] https://www.chef.io

[27] https://Puppet.com

[28] S. Pandey. ”Investigating Community, Reliability and Usability of CFEngine, Chef and Puppet .” Department of Informatics, University of Oslo, 2012.

[29] K. Torberntsson & Y. Rydin. ”A Study of Configuration Management Systems .” Uppsala University, 2014.

[30] T. Delaet, W. Joosen, and B. Vanbrabant. ”A survey of system configuration tools .” LISA. 2010.

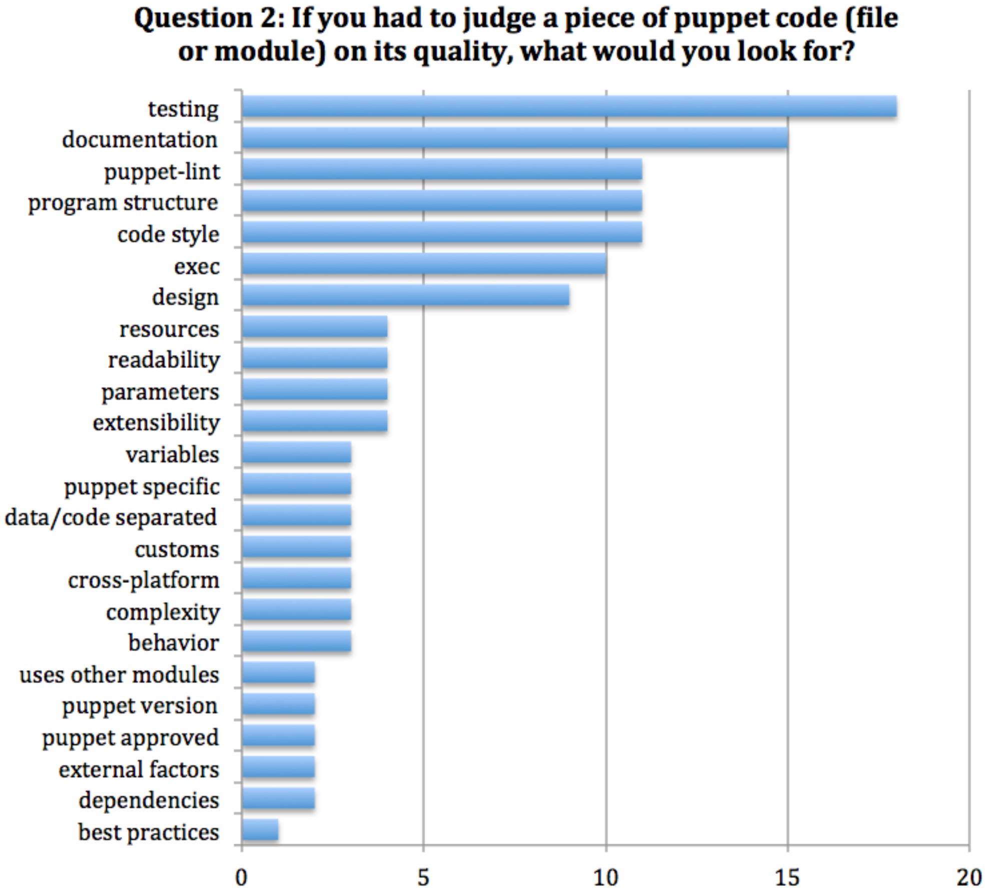
[31] https://docs.puppet.com/puppet/

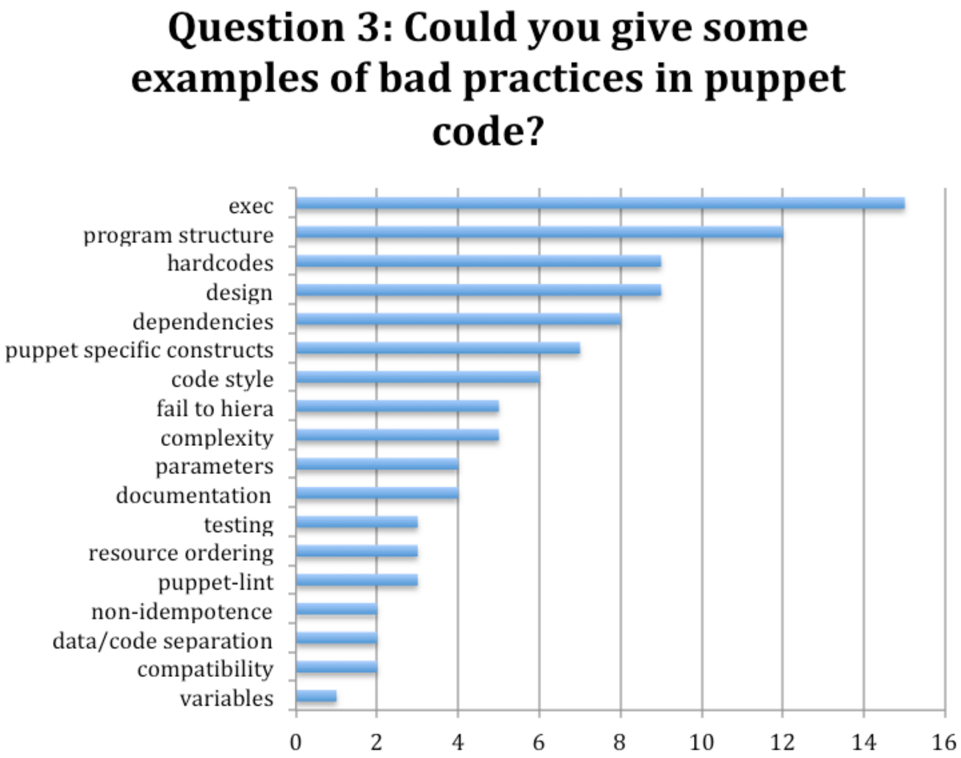
[32] http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey

[33] https://github.com/puppetlabs/puppetlabs-motd

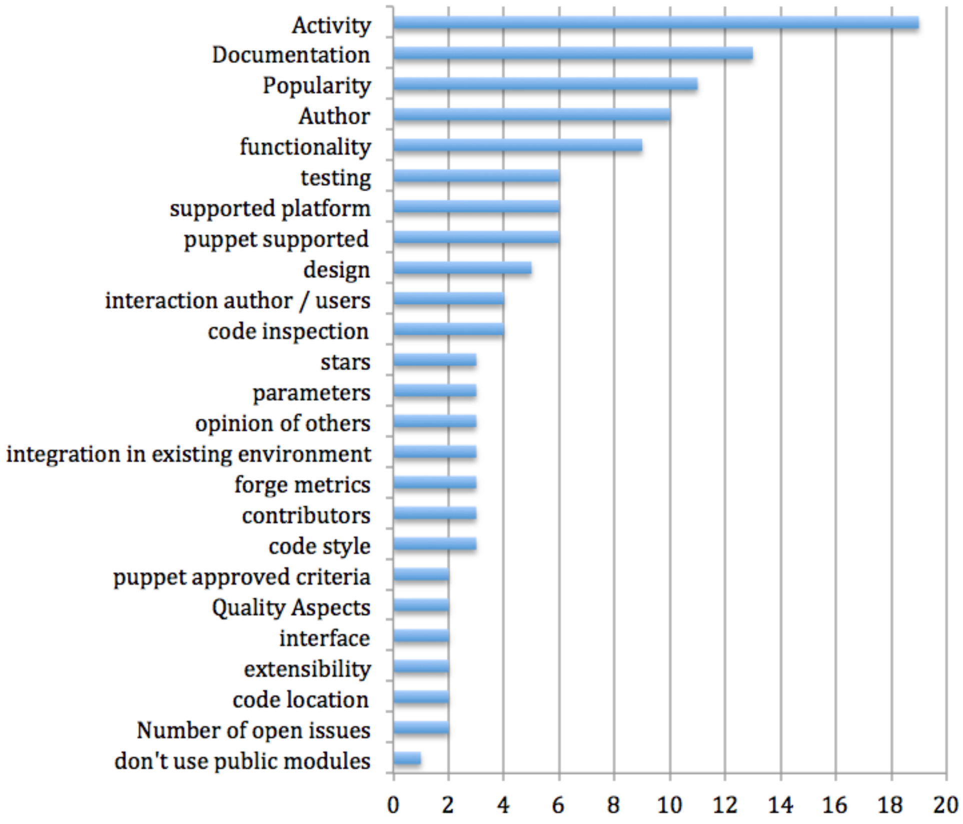
Appendix A

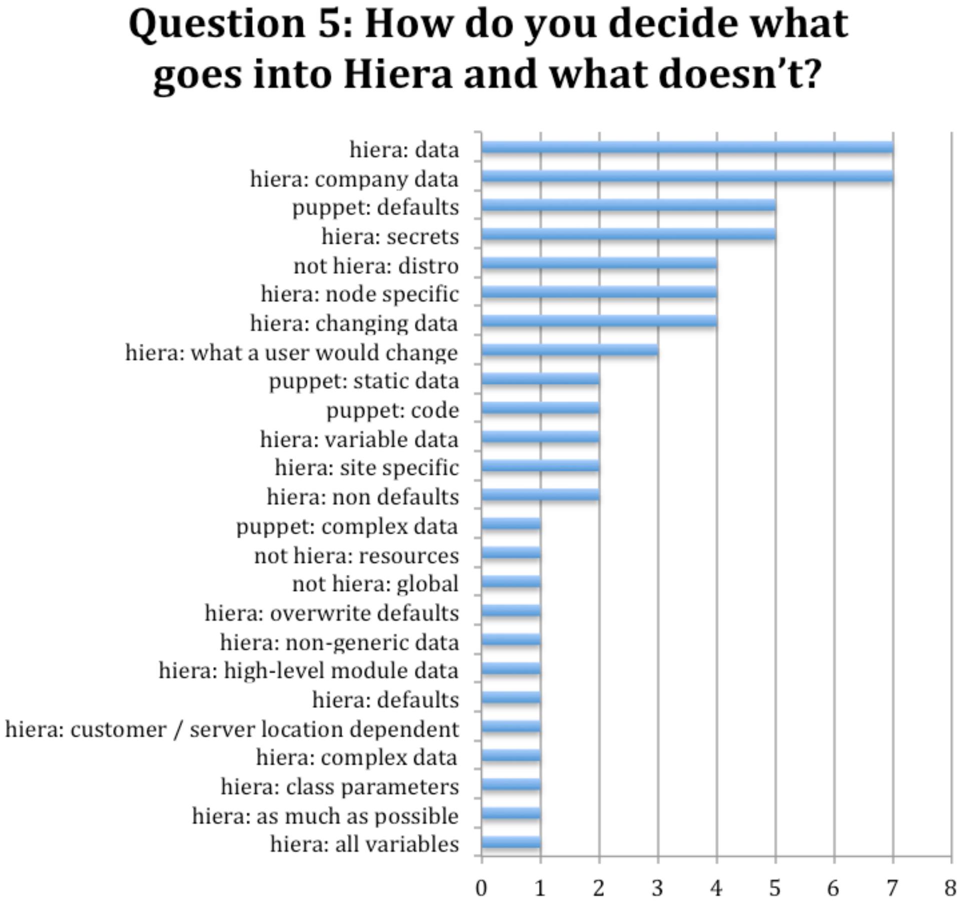
Survey code charts

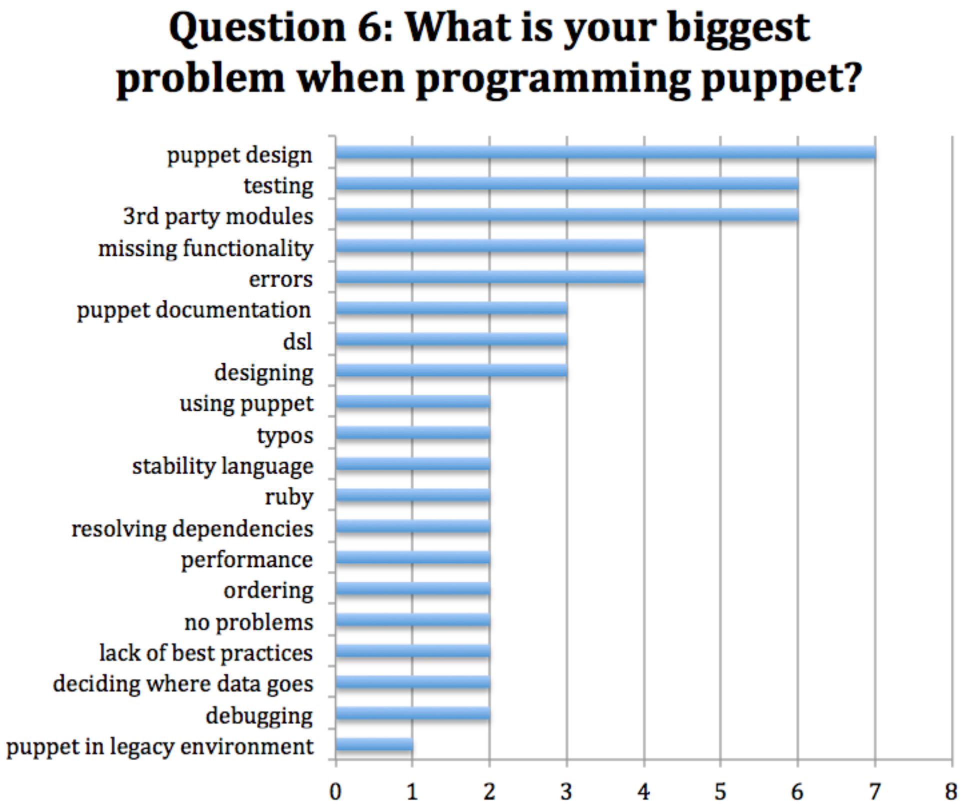



Question 3: Could you give some examples of bad practices in puppet code?
exec
program structure
hardcodes
design
dependencies
puppet specific constructs
code style
fail to hiera
complexity
parameters
documentation
testing
resource ordering
puppet-lint
non-idempotence
data/code separation
compatibility
variables
0
2
4
6
8
10
12
14
16

Question 4: What aspects do you take into account when selecting a puppet module from GitHub or Puppet Forge?




Question 5: How do you decide what goes into Hiera and what doesn't?
hiera: data
hiera: company data
puppet: defaults
hiera: secrets
not hiera: distro
hiera: node specific
hiera: changing data
hiera: what a user would change
puppet: static data
puppet: code
hiera: variable data
hiera: site specific
hiera: non defaults
puppet: complex data
not hiera: resources
not hiera: global
hiera: overwrite defaults
hiera: non-generic data
hiera: high-level module data
hiera: defaults
hiera: customer / server location dependent
hiera: complex data
hiera: class parameters
hiera: as much as possible
hiera: all variables
0
1
2
3
4
5
6
7
8


Question 6: What is your biggest problem when programming puppet?
puppet design
testing
3rd party modules
missing functionality
errors
puppet documentation
dsl
designing
using puppet
typos
stability language
ruby
resolving dependencies
performance
ordering
no problems
lack of best practices
deciding where data goes
debugging
puppet in legacy environment
0
1
2
3
4
5
6
7
8

Appendix B

Interview summaries

B.1 Interview 1

The initial ranking made here was by using shell commands to quickly navigate the repositories, counting Lines of code, lines of commentary, looking for the number of exec's, if's and case statements, parameters, looking for "todo/fixme/hack" comments, a search for hardcoded data (certificate, ssh key, ip-address, etc.). Finally a manual inspection was done on some repositories.

1. Have not encountered problems very often.

2. Execs don't match up with the philosophy of Puppet (they are not declarative). It is an emergency solution. Development is not supported by the IDE, and is for instance comparable to direct SQL queries in Java. For every exec, you have to think about when Puppet has to skip executing it (and write this as well).

3. Lint warnings can be removed. Some teams may not use it or have certain warnings turned off. Adapting a tool into the model is dangerous, for if the tool changes or disappears the model has to be changed as well. Teams may use different tools than Puppet-lint.

4. What should be added is a metric to rate how well a module is structured. In one of the examples, there were sub-sub-sub modules, or a module with way too much files when compared to usual modules that people write (non-third party). A volume to #modules ratio could be considered as well, or the number of .pp files a module has.

5. Yes, I consider them useful with the changes I proposed.

B.2 Interview 2

The initial ranking was done using the tool 'Puppeteer' and calculating the density of design and implementation smells reported. Some random files were opened as well to look at them.

1. No, as I am not a Puppet programmer.

2. Puppet is declarative, and whilst it is sometimes needed, overuse of it makes it that you are no longer following the principles of the language.

3. Duplication may have to be changed, since if there is a lot of boilerplate (i.e. a number of file resources with 10 identical attributes) it would already be detected as duplication.

4. Something to rate the repository structure. I.e. is it only a naked list of modules or is it an actual configuration with a main manifest, modules folder, etc. More design level metrics could be added, such as detecting an empty class/define, or detecting deficient encapsulation within a module.

5. I think the model is definitely useful as an indication of quality. Improving on individual metrics is still a call to be made by the programmer. Sometimes it is not possible to change/improve it, or it comes at a price too heavy to pay. Don't blindly follow the metrics – its still the programmer's decision.

B.3 Interview 3

The interviewee did not have enough time to properly look at the repositories. He mainly looked at the directory structure and a proper readme.

During the rating which took place in the interview, grep commands such as a search for 'password' were done, as well as looking for the largest Puppet files.

1. Maintainability issues: Modules getting interwoven over time. Upgrading third party modules.

2. Yes, overuse is bad. One problem with execs is that always something ends up in the output, but this was not very major. Overall the current use of exec's does not impact the maintainability. Sometimes writing the proper type or resource instead of an exec is simply not worth the investment/trouble.

3. Both lint and too many exec's could be moved into something as 'violations'. Having many parameters is less of an issue in Puppet, so this is not as important. These are things you simply should not have.

4. Tests are missing from the model (i.e. test code volume, assert density). Documentation is missing. Passwords, hardcoded URL's (especially those going to external sites, which means that if they go down, your build fails). They are not always bad (such as relying on Maven), but it is something to take into account for rating a configuration on its quality. Todo's could be added as well. Finally, how do they deal with secret information? Are they using eyaml, or a custom solution? Are they using it consistently?

5. Yes, I think they are useful. Parameters could be changed to required parameters instead of all parameters. The code has a lot easier to use if I can just include a module and it works.

B.4 Interview 4

Took about 15 to 30 minutes to look through the repositories. In making the judgment, he looked for execs, complexity and the filelength.

1. The biggest maintainability issues is exec's that always run. We don't have any problems with the if-pattern (feature creep inside modules). Dividing code in components that make sense is difficult as well. Upgrading third party modules is difficult as well, since sometimes upgrading one module represents a big-bang upgrade. Generally I think Puppet is an easy language but it is still possible to make complete spaghetti code.

2. Very neat Puppet code puts datatypes in a Ruby type or a define. With execs it becomes a shell script inside Puppet. Reading them is difficult. It is very difficult to track down errors related to execs. There is quite a large step between writing exec's and the proper types, which has to happen quite early in the stages of programming Puppet. Puppet is not designed to be used with execs.

3. Lint-warnings could probably be left out. Some warnings are turned off or are unavoidable (this line has to be longer than 80 characters). When I'm using Geppetto I try to remove all warnings that pop up. Duplication is different in Puppet, since it is possible to copypaste something and then rearrange the ordering of attributes, avoiding clone detection but not change its meaning. Generic clone detection won't find this, so a more advanced clone detector would

be useful here. Finally, parameters are usually quite independent of each other [you can't toss them into an object], so the focus here should be on useful defaults.

4. Exec's that always trigger are problematic – you would want to look at the result of the Puppet run, or analyze every exec to find which one are always running.

5. Yes. You can have a lot of metrics, but then won't be useful. It is better to have a few good metrics than thousands of bad ones. An estimation has to be made if improving the metric is worth the effort.

B.5 Interview 5

The interviewee initially looked for documentation and repository structure, and mainly used the provided metrics to make a judgment.

1. No. Sometimes reading code written by a different developer is difficult due to lack of comments explaining what is going on.

2. Yes. Execs can become troublesome when they are very long and complex. In that case it might be better to put it in a shell file and execute that.

3. Lint warnings say the least about quality. From the surface only looking at lint errors, it is impossible to judge if the code was written well. Having a different formatter than lint is also possible.

4. Not really.

5. For companies or teams involving multiple people the code has to be readable. It is very easy to accidentally copypaste or reinvent the wheel if you cannot talk to each other (in the case of open-source projects). For small teams this might be less important. Refactoring time can be an issue. I think the model is useful for improving quality. Use of up-to-date Puppet can be considered as well.

B.6 Interview 6

The interviewee did not look at the repositories beforehand. For every repository he looked at the top-level structure and looked for documentation. Also looked at the main manifest of the repositories, and also looked at random modules to see the code style, which well enough in every repository. Tests were also looked for, and if present, what was tested and how the test was structured.

1. An example is a hobby project of hosting. There were no full system or any tests present. For upgrading to a new version of Debian, the functionality of the entire system had to be reviewed manually. This proved to be too much effort to do.

2. Yes, I agree that it can be overused. Every time you use an exec you do something dangerous. Sometimes that risk has to be taken – it falls outside of the safeguards that Puppet provides to you.

3. –

4. Node Complexity – How much code is applied to a single node? How many types of nodes are there? Try to see which modules are a hotspot of activity. Very low activity might mean less risk for maintenance upgrades.

5. [I asked about the metrics and mentioned Apache as an example (which has large, complex and lots of parameters)] Apache is the goto example of a project that went off the rail. The vhost.pp is very large and still doesn't cover everything a vhost does. Parameters for instance could be reduced by making a template and let that consume input. Values consistent across infrastructure. [Directing the discussion back to usefulness of metrics] You can use metrics to

characterize a problem, and to see how it improves. You might get code coverage to 80 or 90%, but if that that only covers code that is rarely used your coverage is still low. Solid base + good coverage is a good indication. Pared with an understanding that it is not sufficient quality. It is possible to score great on all metrics yet still have bad code. They can also offer guidance on where to look next.

B.7 Interview 7

The interviewee looked for documentation and mostly looked at the repositories with how easy it would be to start maintaining this.

1. Yes and no. Most problems are self-inflicted, sometimes the tooling has issues but we avoid this by staying 1 version behind the newest. Creating tests and keeping up with them is probably the biggest problem.

2. I hate execs. Exec exist outside catalog, so it only reports success or failure. Sometimes there is no type/provider you can use. If you lean on them, you're using Puppet to do the same old. Why then use Puppet? Only use it when appropriate.

3. –

4. Tests. Both the presence of unit & acceptance tests. Acceptance tests are more informational – i.e. the impact on an organization. Documentation, which gives you a long-term ability to add people to the project and give a general understanding.

5. I think a risk profile is great, I'd want to implement this myself. Metrics are great to track, modify and tweak them. I'd be interested to apply this at my workplace.

B.8 Interview 8

This interviewee considered volume as the most important metric.

1. Yes. Deep dependencies of third party code, bad encapsulation. You have to see what happens in the source, it is not really an API you can talk with. Where are modules dependent of each other? Implicit dependencies can be difficult as well. Transitive dependencies. Keeping track of order in an explicit manner is difficult.

2. Yes, it can be overused. They have problems with idempotence. They are custom and not declarative.

3. Filtered lint warnings. There is a possibility that warnings don't matter. They are also treated as all being equally important. Fan in / degree – I would expect fan-out here, since fan-in is less important.

4. -

5. Yes, I think they would be useful.

B.9 Interview 9

This interviewee looked only at volume when ranking the repositories.

1. No. I basically deal with applications and writing modules for them, so I don't have to upgrade or updates. The only maintenance problems that arise are when the underlying API of a module changes.

2. The point of Puppet is to automate. A module should work across multiple operating systems. They are hardcoded, inflexible, contain magic values. Not declarative. Complicated bash/unix is undecipherable, a native type is much more readable.

3. Lint warnings may be split between internal modules and third-party modules. Execs are a good indicator. Complexity inside modules tends to be very simple and should. Complexity should be much lower in configuration you know already. Complexity should be there for different operating systems or machine types. Separate fan-in/fan-out across modules. Would be interesting to see, very hard to generalize to good or bad. I think volume is the most important metric. How longer the manifest, more changes will be made from the defaults specified. More chances of doing something incorrect, such as contradicting attributes. You may set both the memory and connection to X-Large, which can have contradicting effects. Or turn concurrency off and set the poolsize to 100.

4. The number of users that a third-party module has and the number of maintainers could be interesting to spot end-of-life third party modules that are used. Do they use r10k / codemanager? Storing third party modules inside own git is not the Puppet way of versioning. Variable usage? If no variables are used inside something that configures for example a webserver, then everything is hardcoded. Do execs contain variables, or are they flat strings? The number of classes in a module: both public and private classes.

5. Yes, definitely. Compared to both internal and external metrics, this is a better approximation. The metrics are easy to gather and valid.

B.10 Remarks on repositories

Interview	462 - gov_uk	533 - NLNOG-RING	431 fuel_infra	290 operations-Puppet	736 kilo-Puppet
1	High number of “todo/fixme/hack”. Most files and units short. SSH keys in manifests. 1/3 of lines is commentary.	Few “todo/fixme/hack”. All nodes are declared with hard-coded owner and location. Hard-coded ipv6 addresses.	Few “todo/fixme/hack”. Sub-sub-sub classes. Some classes have lots of parameters. Hard-coded certificate.	High number of “todo/fixme/hack”. High number of execs. Large volume.	High number of “todo/fixme/hack”. High number of execs. Large volume. Cases w/ OS-family. Large if-else blocks, lots of comments, classes are called with many parameters set.
2	Few code smells.	High smell density.	Medium smell density.	Lower smell density.	High smell density.
3	Different environments. No eyaml. Unencrypted password found in code. Inconsistent password management. God Module.	Handles passwords better. Graph is visually pleasing. Does have a list with (inherited) nodes, which will give upgrade problems going to Puppet 4.	No Readme. Hardcoded external URL. If !defined() – workaround for Once-and-Only once resource declaration. A lot of ifs (and not for OS-specifics). Ideally you don’t have complexity. God-Module, along with multiple modules with same name.	Very large, tightly coupled. Very difficult to understand if you were not part of development during growth. Adjusted third party code.	No Readme. Adjusted 3rd-party code. No separation of third party and own code.
4		Multiple classes in the same file, which is a Red Flag for me.		Doesn’t have that many smells, uses validate sometimes.	
5	Scores well on all metrics. Some parameters. Very few execs, lots of tests, scores well on Puppet-lint.		Large files.	Metrics look good, but still has tight coupling.	Only has a list of modules. Complex, has a lot of parameters and duplication. Lots of commentary, but not sure if this is bad. No tests.
6	Has a Puppetfile. Has documentation, tests. GOVUK module is a central module. Mature codebase, documentation, also on how to test it, tests in modules	Very small, uses some default modules.	Smaller codebase.	Has a base module, which might be a red flag – Is it code specific to Wikimedia or is it a pile of mud that does everything? Would be my first thing to improve. This core piece of infrastructure does have tests. Documentation on how to write code is present.	Reflects my opinion on openstack, lots of interconnected passages, not very ewll structured. It is based on keystone and mysql. Ceilometer has some dependencies. Nova is connected to everything. This might reflect on OS issues as a whole or that cloud-issue space is a hellishly complicated thing.

Interview	462 - gov_uk	533 - NLNOG-RING	431 fuel_infra	290 operations-Puppet	736 kilo-Puppet
7	Uses Icinga for monitoring, Nginx, has a readme and Jenkins/Travis tests. I can work with this. Has its own rake tests, doesn't use spec helper. Readme per module would be nice. No init.pp in backup. Makes it difficult to start on module. Classes are nicely written, have tests. They are aware that documentation is missing at some points. Use the PCS-pattern. Parameters are documented. The repository is sizeable but manageable. Comfortable, but room for improvement. Very low number of execs, the configuration is composable and understandable.	Random files, bootstrapping? Older codebase, uses import statement. Early Puppet 3/ 2.7. It works. Lots of data in node statements. Uses a 3 year old version of stdlib, which can be confusing for a new programmer who is used to a more recent version. Difficult to figure out what is happening in codebase. Files are spread out. The project is cool, but it is difficult to figure out what they are doing. Low number of execs. Upgrading is very difficult.	Has some python tests, which is weird, don't know what they do. Parameters have helpful defaults. Node definition is not according to best practices. Uses templates, which is nice. Modules only have manifests and templates. This works, but is not ideal. Puppet class can't be included, which means more configuration management. Glad I'm not working with it. No tests, no way to verify code behaves as expected. Refactoring would be done by manually observing things. Uses zabbix. Change in one place could affect multiple nodes. No init in fuel_project, no readme's anywhere. If I had to work with this code, I'd have no idea where to start. Long classes, tons of parameters.	Roles but no profiles. Lot of guides, Puppet-lint/ rubocop. Still use import. Regex in node definition, unusual. Role keyword (profile in node definition). Some conflicts with documentation. Lots of data in code. Great when it works, but when it doesn't. . . . Weird formatting, unsure if compliant. Static data in code, no parameters. Lot of documentation, project is in transition. Code doesn't look bad. Base test looks ok. Pretty big, pretty complex / intimidating. Not sure if they are at fault. The number of parameters is low given that they use Hiera. Upgrading is very difficult.	Lots of interrelated components. Uses openstack/ quickstack/etc. Unclear how things relate if you make a change. No readme. It's all thrown in together. Has empty spec dirs. Quickstack: lot of dependencies, no readme, no tets, custom facts but no documentation for them. No idea what it does. This looks scary. Doesn't follow the DRY principle. Looks like they use Hiera. No tests inside module, no idea if it works now. Lots of forge modules, but not tracked as such. Not worst in the world, but there is lots of room for improvement. I'd go crazy if this was my personal project. Long classes, tons of parameters.
8	Short files, small modules. Well structured.	Quite small.	Own script to install and upgrade third party modules.	Difficult to maintain, largest repository.	No separation of third party code and own code. High ratio of LOC vs modules. Only has top-level modules. Code does look good. A lot of class resources in quickstack.
9	More own modules than the others	Very Simple	#2 in size	Very large, lot of modules, lot of stuff to configure – more parts that talk with each other increases complexity	Lot of third-party modules, so less to maintain