Usability of System Configuration Languages:

Confusion caused by Ordering

Adele Mikoliunaite

Master of Science

Cognitive Science

School of Informatics

University of Edinburgh

2016

Abstract

In recent years the demand for IT services has outgrown the supply of skilled system administrators. Such changes resulted in the system administrator group shifting from a predominantly experienced and trained pool of resources to include novice and self-taught administrators.

Setting up services, installing software and troubleshooting in a constantly high pressure environment while lacking extensive knowledge of the system and having to use tools designed for the experienced only, can be not just challenging for a system administrator, but also expensive for the company when incidents arise due to the combination of these new factors. Due to massive outages in large companies across the world, caused by configuration errors, an increasing amount of research is being conducted, aiming to prevent system administrator mistakes. Such mistakes are caused by misinterpretation or lack of knowledge, and since declarative configuration languages, are often thought of as more complex to get used to for novice administrators, we have studied three of them – Puppet, SmartFrog and L3.

To assist the research we have built a survey to test intuitive judgement and preferred interpretations of such features as *referencing*, *inheritance*, *scope* and *ordering* presented in pseudo code. It is thought that order within declarative languages is not important, however collected data shows that ordering, in fact, does matter in certain contexts and especially when paired with other features such as inheritance or referencing.

Acknowledgments

I would like to thank my supervisor Paul Anderson for proposing such an interesting project, accepting me for it despite my lack of knowledge in the field and continuing to guide me into the light throughout the lifespan of this project. I would also like to thank my second supervisor Kami Vaniea for always finding time for me and my questions in her very busy schedule and all the detailed comments that have taught me a lot. I would like to thank Hamish Macleod for the discussion we had and friendly help in navigating statistics. I would also like to thank Ken Barber, Jenny Mahmoudi, Nigel Kersten, Tom Limoncelli, Eser Kandogan and Ski Kacorski for their positive support and promoting our survey. I would also like to thank to everyone who answered the survey.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Adele Mikoliunaite*)

Table of contents

ABSTRACT **3**

ACKNOWLEDGMENTS **4**

DECLARATION **5**

TABLE OF CONTENTS **7**

1. INTRODUCTION **9**

1.1 CURRENT SOLUTIONS 11

1.2 OUR APPROACH 12

1.3 RESULTS 14

1.4 STRUCTURE OF THE DISSERTATION 15

2. BACKGROUND **17**

2.1. SYSTEM CONFIGURATION 17

2.1.1. Configuration management 17

2.1.2. Declarative approach to configuration management 20

2.1.3. Configuration languages 21

2.1.4. Order matters 22

2.2. HUMAN FACTORS OF SYSTEM ADMINISTRATION 24

2.2.1. System administrators 24

2.2.2. Configuration errors 26

2.2.3. Human Error 27

2.2.4. Mental models 29

3. CONCEPT DEVELOPMENT **31**

3.1. INHERITANCE 32

3.2. SCOPE 36

3.3. REFERENCING 37

3.4. ORDERING 39

3.4.1. Evaluation order 39

3.4.2. Lexical order 41

4. METHODOLOGY **43**

4.1. CHOOSING THE METHOD 43

4.2. DESIGNING THE SURVEY 44

4.3. PILOT TEST 45

4.4. IMPLEMENTATION 46
4.5. DATA ANALYSIS 48

5. RESULTS 51

5.1 DATA DESCRIPTION 51
5.2 CONFIDENCE LEVELS 53
5.3 MENTAL MODELS OF CONCEPTS 55
5.4 CONFUSION OVER ORDERING 61

6. DISCUSSION 65

6.1 CONCLUSION 67
6.2 LIMITATIONS 68
6.3 FUTURE WORK 69

BIBLIOGRAPHY: 71

APPENDIX A 77

1. Introduction

In recent years, attention to configuration errors has increased vastly. This can be directly attributed to the number of major system failures around the globe caused by misconfigurations. In 2009, incorrect configuration brought down the entire .se domain in Sweden (CircleID.com, 2009), which effectively stopped the Swedish National Internet from working for an hour. In 2010 a configuration error took Facebook down for 2.5 hours, resulting in the worst outage at Facebook in over four years (Johnson, 2010). In 2012, the misconfiguration at Amazon rendered Netflix unavailable on Christmas (The Aws Team, 2012). Since then, other large outages, caused by misconfigurations, were experienced by Amazon (The Aws Team, 2011), Microsoft (Miller, 2012; Sverdlik, 2014), Google (Brodkin, 2012; The Google Apps Team, 2013), LinkedIn (Liang, 2013), and Home Box Office Inc (York. 2015). Such configuration errors impact customers at various levels and cost companies not just the income lost from sales, services and reputational damage, but also the time of employees (Patterson, 2002). The results of misconfigurations can be irreversible, especially if important data was lost.

In this work, *system configuration*, refers to the process of installing software packages, and setting and managing system parameters, based on requirements (e.g. increasing security, improving performance, allowing certain functions). System configuration could be used for managing one computer to managing hundreds of servers in large scale organizations and for operating large and complex systems across multiple machines. In such situations, system configuration can be extremely complex, consisting of thousands of parameters, interacting in various ways (Xu & Zhou, 2015). Such complexity makes it extremely difficult to prevent, detect and troubleshoot configuration errors.

Configuration error, here, refers to the unsatisfactory behaviour of the system, which could be caused by *misconfigurations* (incorrect settings of the system). Configuration errors may be divided in two types: *system errors* (e.g. corrupted data, software bugs) and *human errors* – system administrator mistakes (Xu & Zhou, 2015). Some human errors could be due to *action slips* (e.g. typos), like a missing dot, which was identified as the root cause for the .se domain outage in Sweden (McNamara, 2009), mentioned previously. Other human errors can be due to *mistakes,* which refer to misinterpretation of the system. The configuration error study by Yin et al. (2011) showed that the biggest part of misconfigurations (70.0% – 85.5%) were made up of configuration *parameter* mistakes. They divided parameter mistakes into *illegal* and *legal*. Illegal parameter mistakes are the setting of values that violate format, syntax or semantic rules, while legal parameter mistakes are setting incorrect values, that do not violate any rules but they do not deliver the performance intended by the user. The latter had been found to be the case in up to 61% of mistakes within parameter configurations by Yin et al (2011). Xu et al. (2016) discovered that contrary to expectation, more than half of misconfigurations made by humans are a result of an administrator's difficulties in finding the right parameters and setting correct values, i.e. *mistakes*, not the *action slips*.

The reason behind these mistakes could be a recent shift in the system administrators group. System administrators used to be experienced professionals within their field, but due to the rapid advance in IT services, the demand exceeded the supply. Today the majority of system administrators enter the field without having received any formal training. At best, system administrators have a degree in computer science or are self-taught, since education or preparation in system configuration, specifically, is not yet widely available (Border & Begnum, 2014). In addition, system administrators are very different to the software developers or end users (typical computer users), in a way that they do not have extensive understanding of controlling and debugging systems, do not write code (and often cannot read it) and mostly depend on user manuals, colleagues advice and Internet sources to perform configurations and understand systems (Xu and Zhou, 2015). Thus, system administrators vary from highly skilled and experienced professionals, in large organizations, to novice administrators, lacking both skill and experience. In contrast most tools and user manuals are created for people with some experience and knowledge, not keeping the novice users in mind. Such a change in the group of users of these tools and such a high rate of configuration errors indicates that it is time for tools to change accordingly to accommodate the shift.

1.1 Current solutions

There have been many attempts to build various automated tools to help diagnose and fix configuration errors. Keller, Pupadhyaya and Candea (2008) presented a tool *ConfErr* to test how resilient to human errors software is. It was used to generate human like configuration mistakes and test how systems are able to handle these errors. It then produced a resilience profile, which could be used as a guide to improve the software. The purpose of dynamic flow analysis tools, like *ConfAid* (Attariyan and Flinn, 2010) was to find root causes of misconfigurations by monitoring causal dependencies as the code executes and linking them to corresponding tokens in configuration files. Static analysis tools, such as *ConfDebugger* (Dong, Ghanavati and Andrzejak, 2013), were built to capture lexical and syntactic errors (*slips*) in code without executing it. All of these and many other tools can successfully detect most mistyped configuration and other human *slips* such as typos. However, in dealing with human *mistakes* and misinterpretations automatic tools might not be the best approach to utilise, as mistakes are normally legal values that conform to syntax rules but were chosen incorrectly for the goal intended.

Due to the mentioned changes in system administrators group, Xu, Pandey and Klemmer (2016) argued that their errors should be treated as Human-Computer Interaction problem. In recent years various aspects of user experience (e.g. end-users) have received a great deal of attention and investment from Industry, but system administrators have been overlooked in this context (Oppenheimer, Ganapathi and Patterson, 2003). It is easy to think of system administrators as the handymen of the IT Industry, who know it all, but at the end of the day they are also users, who depend highly on their tools. System administrators spend significant amounts of time using various tools, which were often developed in an ad hoc manner for people with skill and experience. They work under constant pressure managing hundreds of configurations and troubleshooting issues with tools that do not support their needs. They are required to collaborate successfully, keep their focus at all times and create workarounds to compensate for the deficiency of the tools provided (Barret et al., 2004), so that everyone else on a workstation can do their job. It would be unjust to consider configuration errors outside of the context in which they appeared. The underlying issue of misconfigurations, as argued Xu et al. (2013) might as well be the system itself, as well as configuration design. Or as Norman (2013, p. 162) claimed:

> “Most industrial accidents are caused by human error: estimates range between 75 and 95 percent. How is it that so many people are so incompetent? Answer: They aren’t. It’s a design problem.”

Nagaraja et al. (2004) created a validation technique to catch all the operator mistakes in a virtual environment before making it visible to the system and users. Their prototype was proved to find two thirds of the 42 mistakes they were testing it on. A slightly different approach was taken by Xu et al. Arguing that rethinking and redesigning configuration to prevent users’ confusion, next to auto-configuration is a part of the ultimate solution against configuration errors, Xu et al. (2013) have created a tool called SPEX to automatically infer configuration requirements from the source code of the system in order to expose vulnerabilities and error-prone configuration design.

1.2 Our approach

In order to prevent errors through design changes, it is first needed to understand what exact issues the current design holds, what are the aspects that confuse users the most and leads them to make mistakes. Xu, Pandey and Klemmer (2016) took a step in this direction and suggested that in order to help system administrators we should start by understanding their cognitive problems – which tasks are found to be difficult and why? However, the underlying problem Xu, Pandey and Klemmer have not discussed is that system administrators, especially novice, are not always aware of their misinterpretation of system behaviour, especially if nothing results in an error immediately. Therefore, they might not think that something was difficult or that they had an issue, when in fact they did. Such constraints require different methods of research to be applied.

Oppenheimer, Ganapathi and Patterson (2003) stressed the importance of a match between the model of the system and the mental model system administrators form about the system. *Mental model* in this study refers to an internal representation (in human mind) of a system or a concept. Hrebec and Stiber, in 2001 tried to follow this direction to discover what mental models system administrators had regarding the systems they worked with. However, their survey was not successful as the number of participants and the information they gathered did not provide much insight about the mental models system administrators had. In particular, there was no data indicating or at least hinting as to the structure or behaviour of the system. Instead, it explained how difficult they thought the systems were and revealed how they were approaching issues in the work place (e.g. by experimenting, asking for help of someone you trust, etc.).

In this study we focus on custom configuration languages of such declarative configuration frameworks as Puppet, SmartFrog and L3. *Configuration language* refers to a configuration specification written in a higher level and *declarative approach* means defining configuration in terms of *what* needs to be done, as opposed to *how,* used in imperative approaches. Declarative configuration frameworks were chosen due to their growing popularity and complexity. In addition, it is likely that declarative tools are more difficult for novice users to get used to, since most of programming and scripting languages are imperative. Also, since it is often stated that order does not matter in declarative languages, we were interested to see whether it is a real feature that system administrators are also aware of, or a source of confusion instead. Finally, our checks against existing research have not come across previously conducted analysis of usability of configuration languages. Therefore, we chose to concentrate on ordering within the mentioned configuration languages as a manageable problem.

The purpose of this study is to research the confusion, intuitive judgement and expectations of the actual and potential configuration language user, in order to gain better understanding of the usability of configuration languages, impact of ordering and how it relates to configuration *mistakes*. A survey was conducted to find out what expectations towards configuration language features such as referencing, inheritance, scoping and ordering system administrators have and on what mental models they tend to rely on, when presented with unfamiliar pseudo configuration examples. The underlying goal and our main objective is to detect the weak spots within these features that cause misinterpretation and, therefore, mistakes. By uncovering these weak aspects within configuration languages we hope to help to improve existing and create better configuration languages, suitable to use by both experienced and novice system administrators.

Qualitative data analysis was used to study the underlying models of chosen declarative frameworks. The particularly confusing and self-contained feature – referencing (including inheritance and scoping) was chosen to explore in more depth. Descriptive and exploratory analysis was used to process the gathered data.

1.3 Results

The survey has been completed by 384 participants. Majority of the participants were from IT related background and had experience working with configuration languages. Many were familiar with configuration tools such as Puppet and Ansible. Participants had high confidence levels and good skills, as measured by the survey.

Survey results revealed that the highest rates of confusion lies within the inheritance and ordering. On the surface it appeared that most participants were familiar with the inheritance, but occasionally they had picked answers that did not fit into the same model of inheritance that they followed on in different questions. Multilevel inheritance was supported in two different ways by more than half participants, although it also had a group of people who preferred multilevel inheritance not to be allowed, as well as a group of people that did not know how they would expect it to behave.

Nearly all participants supported dynamic scope for classes, however a clear division between static and dynamic variable scoping could be seen. In relation to this, future referencing was only partially supported by half of participants, while slightly smaller group preferred thorough dynamic approach with future references supported for both variables and classes.

1.4 Structure of the dissertation

The structure of the dissertation is as follows:

Chapter 2 explains declarative approach to configuration management, introduces configuration languages, discusses why misconfiguration happen and presents Human centred approach in targeting misconfigurations.

Chapter 3 presents work undertaken, including concept and survey development, explains chosen concepts in detail, and discusses final design of the survey and lessons learned through testing.

Chapter 4 presents the results of the survey, describes the qualitative data and discusses most significant findings.

Chapter 5 concludes the work, discusses findings and limitations, and presents suggestions for future work.

2. Background

This chapter explains what is system configuration, who are system administrators, why do they use configuration languages and why usability of it matters. The first part of the chapter focuses on system configuration used in IT management. It explains the difference between imperative and declarative approaches to system configuration and introduces configuration languages. Advantages and disadvantages of declarative configuration languages are discussed and reasons behind confusion over ordering are reviewed. The second part focuses on human factors of system administration. System administrators experience, workspace and tools are discussed as well as the requirements they are expected to meet.

2.1. System configuration

2.1.1. Configuration management

Configuration management is used in IT service management, military, civil engineering and industrial engineering. Since the use in each field is different, this paper only discusses configuration management in the context of IT service management. In the context of large sites, for example, for thousands upon thousands of Google developer workstations, configuration management would be used to make sure the workstations are functioning well and accommodate developer’s needs. Meaning, their computers are running smoothly, allow needed access and support required software packages. The network needs to be up at all times, avoiding disruption of workflow and losing any data. For the amount of emails generated in such large sites, mail servers have to be well managed as well. *Configuration management* is used for such purposes amongst others.

The term *configuration* derives from Latin *com-* “together” (“with”) and *figurare*, “to form”, which together means “to form from or after”. Another meaning is “a relative arrangement of parts or elements”, thus *configuration management* means managing an arrangement of parts or elements (Mette and Hass, 2002). Configuration management has many definitions, this study uses one by Mette and Hass (2002, p. 3):

> Configuration management is unique identification, controlled storage, change control, and status reporting of selected intermediate work products, product components, and products during the life of a system.

Anderson (2006) described a few types of configuration. *System configuration* refers to installing software packages on a large number of machines and specifying their functionality based on requirements. It also includes reconfiguration if and when needed. For example, when there is a change in environment, specifications or when something breaks. *Hardware configuration* refers to building overall systems from hardware parts, based on requirements. *Software configuration management* involves putting modules together to construct complete software applications. *Network configuration management* refers to configuration of routers, switches and other network devices. *Distributed application configuration* refers to creating single distributed applications, which runs on multiple computers within the network, by configuring and deploying processes onto different computer nodes. *Per-user configuration* involves configuring application settings according to individual requirements of the user.

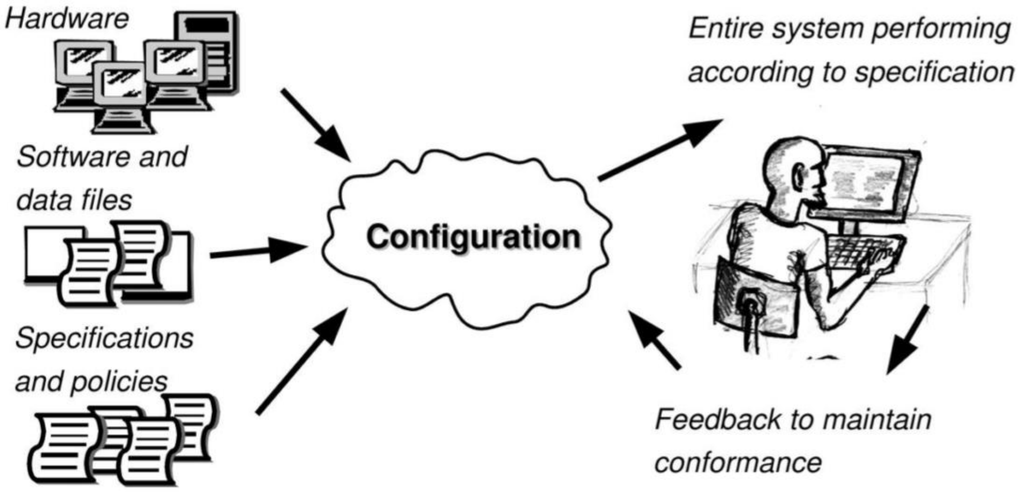

Hardware
Software and data files
Specifications and policies
Configuration
Entire system performing according to specification
Feedback to maintain conformance

Figure 1. The basic task of system configuration (Anderson, 2006, Figure 1).

System configuration management involves features and tasks from the other mentioned configurations. *Figure 1* presents a visualisation of a basic task.

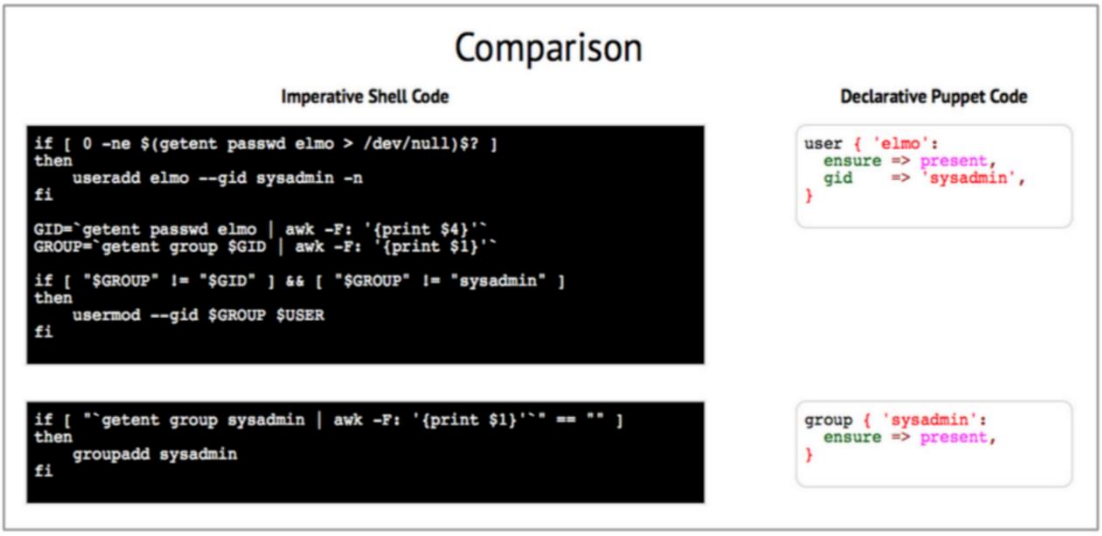
Generally, system configuration involves dealing with a lot of specifications, because all the machines in the site are unlikely to be identical, in terms of what software they contain and what accesses are allowed. *Configuration specifications* define the required behaviour of machines and services in the particular system. They come in the form of settings or parameters, which are stored in *configuration files*. A single configuration file can have thousands, or hundreds of thousands of different parameters. Managing relationships of the machines across the site requires good knowledge of all configurable entities involved. For example, providing Web services would involve configuring not only the Web server, but also DNS and firewall (Anderson, 2006). Large sites usually have separate departments for each type of configuration (e.g. network, mail, firewall, etc.) and many system administrators responsible for configuring different aspects. Configuration languages are used for this purpose, as they are intended to help compose different aspects of the system. In the mentioned example, different people would be involved in configuring the Web server and firewall, and as a result, overall configuration would be composed from different aspects, configured by different people. While in smaller sites the number of people would make it possible to discuss things in person, in large sites this not an option. However, collaboration is a big part of the process. Even though each person involved is expected to have enough skill and knowledge to make their decisions considering the whole picture, the whole picture often is simply too big. Therefore, configuration stands on successful task distribution and collaboration. Each aspect of configuration can have an impact to many configuration files, as well as each configuration file being dependent on many aspects. On top of that, different configuration languages might be used to configure different parts of the system. Such intertwined dependencies are prone to error and especially complex to detangle in the search of the root cause (Anderson, 2006).

Configuration management used to be all manual, but these days there are many tools that help to automate the process to some extent. Some of the well-known tools include CFEngine, Puppet, Salt, Ansible, Chef. They are often written in common scripting languages, such as shell, Python, Ruby or Perl.

2.1.2. Declarative approach to configuration management

There are several approaches to configuration management, but the two most popular are *imperative* (also called procedural) and *declarative*. Following the *imperative approach*, each step that needs to be taken to reach the end goal has to be defined, including all the alternative steps. Essentially, it is a definition of *How to* do things, and it is up to the person writing the script to decide on that. Following the *declarative approach*, the desired final state of the machine is defined, which is essentially the instruction on *What to* do, not how. The tool then computes what steps and in what order these need to be taken to reach the end goal (i.e. *How*). These descriptions, explaining what needs to be done at a higher level are called *configuration languages* (discussed in the next section).

The difference between applications utilising each of these approaches may be seen in Figure 2. Below, by Ford (2014). Both, imperative Shell code and declarative Puppet code are aiming for the same final result, the only difference is how it is defined.



Comparison

Imperative Shell Code

```
if [ 0 -ne $(getent passwd elmo > /dev/null)$? ]
then
    useradd elmo --gid sysadmin -n
fi

GID=`getent passwd elmo | awk -F: '{print $4}'`
GROUP=`getent group $GID | awk -F: '{print $1}'`

if [ "$GROUP" != "$GID" ] && [ "$GROUP" != "sysadmin" ]
then
    usermod --gid $GROUP $USER
fi
```

```
if [ "`getent group sysadmin | awk -F: '{print $1}'`" == "" ]
then
    groupadd sysadmin
fi
```

Declarative Puppet Code

```
user { 'elmo':
  ensure => present,
  gid    => 'sysadmin',
}
```

```
group { 'sysadmin':
  ensure => present,
}
```

Figure 2. Comparison between imperative and declarative configuration code (Ford, 2014).

The *Declarative approach* might seem to produce easier and more concise code. Indeed, it does have some benefits over the *imperative approach*. Ford (2014) claims its repeatability and consistency to be two of the biggest advantages together with the time saved on script writing. Anderson (2006) noted that it allows system administrators to focus more on the end result without being confused over procedures.

It also allows the tools to run continuously monitoring the state and change it whenever it does not match the desired state. In addition, it allows any conflicts to be detected before applying configuration. However, it does present a learning curve for system administrators, especially because they are mostly used to procedural implementations.

2.1.3. Configuration languages

Configuration language refers to configuration specifications, written in a higher level. It is like an abstract description of how systems should look like, it can also include definitions of relationships between entities. Following a declarative approach, configuration languages would contain specifications of what needs to be done, not how. Such high level descriptions allows it to be reused for various goals across different operating systems.

The main difference between *configuration languages* and other programming languages is that configuration languages are descriptive (Weilli, Cheney and Anderson, 2016). Most programming languages widely known today are procedural (C, C++, C#, Java, Python, Ruby, Perl, MATLAB etc.), they use conditional statements (e.g. *if-else*) and loops (e.g. *for, while*) in some form. These features make it easier to define processes step by step. *Declarative configuration languages,* on the other hand, define the end result of the process. Therefore, they use the same and different features to define relationships between entities or specify order, where it matters.

Most configuration tools sit in the range between imperative and declarative, incorporating elements from both. Chef, for example, is at the imperative end of this range, while Puppet is at the declarative end of this range. Imperative configuration frameworks, such as Chef and Ansible usually use already existing scripting languages like Ruby, Perl, Python and bash for their configurations. Even though there is an example of general declarative programming language – Prolog (Anderson, 2006), most declarative frameworks, such as Puppet, SmartFrog, L3 and LCFG have built their own configuration languages (although Puppet is also available using Ruby DSL) to match the requirements. Kanies (2012) argued that the main benefits of the Puppet native configuration language is the simpler and more powerful code. Even though declarative configuration languages follow the same approach, they were built with a purpose to fit entirely different frameworks, therefore, they differ not only in syntax, but also in some of the features.

Declarative configuration languages are mostly being developed in an ad hoc manner by many people in many different places working on different aspects of it. This allows for faster and more efficient development and improvement. However, as functionality of the language grows, so does the complexity of it. Every new aspect introduced into the whole will impact and be impacted by other aspects of it. As a result, many relationships and dependencies will be formed between certain aspects of the language, not all of which will be discovered through testing. Such hidden relationships might also never cause any errors directly, instead, it might appear to work fine, but will produce different results. An experienced and skilled user might discover such dependencies (and will be aware of other ones) and will be able to use some creative workarounds to make it work. While novice user might get overwhelmed with confusion or will not even notice it until it generates an issue (misconfiguration) later on.

2.1.4. Order matters

As mentioned in the previous section, one such confusion, caused by hidden dependencies lies in ordering. It is often thought that in declarative languages order does not matter. However, we found that in declarative languages order does tend to matter in certain cases.

In declarative configuration languages there are a few types of order. Firstly, there is *lexical order*, in which the configuration is written. In imperative language this order is very important, since steps gets deployed following the specifications presented in lexical order. In declarative languages, however, order is often stated as unimportant, since the specifications do not necessarily get deployed in that particular order. *Order of evaluation* is the order in which given specifications are applied. In imperative languages both lexical and evaluation orders are mostly in line. In declarative languages they are not aligned. Furthermore, the lexical order is said to be unimportant and evaluation order is hidden and not in control of the user, as it is decided by the tool/framework. Not knowing the order in which configuration gets evaluated might be a useful feature as it protects users from focusing on elements other than those that are essential. In contrast, it might not be as useful when looking for a root cause of errors.

As mentioned, configuration languages define the desired state in declarative approach, not the steps to get there (imperative approach). Therefore, the tool chooses the best and most efficient order of execution to reach the desire state. The user is unaware of evaluation order, which could potentially cause confusion and mistakes.

In addition, in some cases, different components need to be executed in the specific order to produce required result, and mixing this order up can result in unexpected outcome. Collard et al. (2015, p. 4) presents such example written in Puppet:

```
package{’golang-go’: ensure => present }

package{’perl’: ensure => absent}
```

The desired goal in this example is to install the Go compiler and remove Perl. However, it is not obvious to the user which order will be computed as best by the configuration management tool used and there is no way to find it out without actually executing the code. In addition, the Go compiler is unexpectedly dependent on Perl. Meaning that if Go is installed before removing Perl, once Perl gets removed, the Go will be removed as well. In contrast, if Perl is removed first, the Go will be removed with it, and when Go is installed, the Perl will be installed with it (Mikoliunaite, 2016).

Declarative configuration languages usually have certain features to allow for definition of dependencies when needed. We discuss ordering in more detail in section 3.1.4.

2.2. Human factors of system administration

2.2.1. System administrators

System administrators (also called *sysadmins*) have many different titles such as *database administrator, web administrator, network administrator, storage administrator*, *systems engineer*, *systems architect*, *infrastructure architect*, *system security expert*, *systems operators, webmaster* and other. Just as their titles, their roles also differ. Large companies might have separate departments within their IT department for database, web, network and other servers, while the system administrator in a small company might be responsible for managing all of it. However small or large, each company has at least one system administrator and it is this person who sets up a network within the organization, installs and configures all the needed hardware and software, performs audits for software and systems, applies operating system updates and configuration changes, documents the configuration of the system and ensures that all the services are up and running so that everyone else in the company can do their job. In addition to that, system administrators are constantly diagnosing and fixing all the reported issues, ranging from error message on someone's computer screen to complete server outages. Since the results of outages are often irreversible and can mean not just loss of money, time and customer trust, but also important data, the system administrator's job is crucial to prevent such incidents.

Due to a rapid increase in IT service delivery, the number of servers within companies has accelerated and so has the demand for IT managers (Kandogan, Maglio, Haber and Bailey, 2012). As a result, the costs of the human workforce has outgrown the cost of technology (Bozman and Perry, 2010). These changes encouraged researchers to focus on system administrators in order to find ways to reverse this trend. In addition, such high demand of IT experts resulted in the system administrators group shifting from highly skilled and experienced to include novice administrators (Xu and Zhou, 2015), who at best were self-taught or holding a degree in the field of, or related to, computer science, since education or training in system configuration, specifically, were not widely available (Border & Begnum, 2014).

Since the work of system administrators is essential for any business and due to the change in skill and experience, the need to prevent errors advanced. To understand who system administrators are, what do they do, what problems do they face and what can be done to improve their efficiency and reduce the risk of mistakes, several field studies have been conducted (Barret et al., 2004; Velasquez, Weisband and Durcikova, 2008; Kandogan, Maglio, Haber and Bailey, 2012). Some focused on finding out what tasks of sysadmins could be automated to reduce costs (Kandogan, Maglio, Haber and Bailey, 2012). Others targeted tasks which system administrators found most difficult and explored how they handled their own mistakes through usability studies (Nagaraja et al., 2004). Surveys were used to find out whether the ways systems work matched the system administrators' mental model of it (Hebrec and Stiber, 2001). Barret et al. (2004) conducted several field studies in large universities and enterprises. Through observations, interviews, surveys and collecting artefacts (such as diaries and planning documents) they proved that the tools used by system administrators were often not aligned to their work.

One of the system administrator's daily tasks is to configure applications, server processes and operating systems files, which as mentioned previously, are a collection of settings and parameters that describes the desired state of each. It might not appear complex at first, but Ko et al. argued, that (2011, p. 7):

> "Modern IT systems have hundreds or thousands of configuration parameters that may interact in unexpected ways."

Some applications have graphical interfaces and provide tools to design and modify the syntax of their configuration files. Others may require design and modification of files using a text editor. Skilled system administrators are required to know at least a few scripting languages such as *shell*, *powershell*, *Perl*, *Ruby* or *Python*. However, knowing a scripting language is very different from actually being a software developer. Xu, Pandey and Klemmer (2016) compared user ratings on two Q & A sites, both of which are parts of websites from Stack Exchange Network: StackOverflow.com aimed for developers and Serverfault.com aimed for system administrators. The users in each website are ranked based on their reputation and activity on the sites. Meaning, if the person actively shares his knowledge and other people find this helpful, the person will have a high score. The scores of the users, who were registered at both sites were compared and it appeared that most users who had high reputation in one of the sites, had low reputation on the other (Xu, Pandey & Klemmer, 2016). Such results prove that indeed the skill and knowledge of system administrator and developer differs significantly. System administrators vary from highly skilled and experienced professionals, able to code in variety of scripting and also programming languages to novice administrators, who do not know how to write or read code. However, most user manuals are created for users with some experience and knowledge, not keeping the novice users in mind.

2.2.2. Configuration errors

In 1986, Gray found that administrator (operator) error was the largest cause of failure in deployed Tandem systems, making up to 42% of all failures. He found software bugs to be only the second largest cause of all failures, making up to 25% (Grey, 1986). The later published studies kept on delivering similar results.

In 2002, Patterson conducted two surveys to find that majority of downtime causes were due to system operators (Patterson, 2002). In 2003, Oppenheimer, Ganapathi and Patterson published a study on over 500 failures in three large Internet services: Online (online service/Internet portal), Content (global content hosting service) and ReadMostly (read-mostly internet service) and found operator errors to be the largest cause in two of the three service. More than 50% and in one case nearly 100% of the operator caused failures were configuration errors (Oppenheimer, Ganapathi, Patterson, 2003). In addition, Oppenheimer, Ganapathi and Patterson (2003) discovered that operator errors took more time to repair than other failures.

In 2004 Nagaraja et al. conducted 43 experiments with operators ranging from novice to experienced. The results agreed with previous findings – configuration mistakes were the most common, followed by incorrect software restarts (Nagaraja et al., 2004). Differently from previous studies, Nagaraja et al. took a first step to study configuration errors in the context. They have given tasks to 11 participants and then have observed them completing it and handling their own mistakes. However, the study conducted was very small with only 42 errors from 11 volunteer participants observed and discussed.

Mahajan, Wetherall and Anderson (2002) did a quantitative study of Border Gateway Protocol (BGP) misconfigurations. They focused on such configuration errors as slips and mistakes. In their terms, slips referred to adverted errors whilst mistakes were classified as design errors. They were expecting that most misconfigurations would appear to be due to slips. Therefore Mahajan, Wetherall and Anderson (2002) were surprise to find that mistakes, in particular poor understanding of router command semantics, were the cause for most misconfigurations.

2.2.3. Human Error

Errors, made by people, are called *human errors*. Reason (2000) described two main views of human error: person and system approach. The *Person approach* is mostly directed at high risk occupations, such as surgeons, nurses, aesthetics, and pharmacists, however, it could also apply to sysadmins. It considers acts like forgetfulness, carelessness, inability to focus or lack of motivation to be the root causes of errors. In contrast, *system approach* accepts human error as inevitable, no matter the skill or organization. It views errors as consequences, not causes, of failed defences, and encourages instead of trying to change human nature in making errors, to change the systems and conditions people work in to accommodate for these errors (Reason, 2000).

The term *human error* itself incorrectly implies that it is solely human fault. However, the context in which error happened is as important, especially since many times it contains tight deadlines and limited resources (Ritter, Bacter and Churchill, 2014). As mention previously in the section 2.2.1, system administrators often work in a highly stressful environment, equipped with insufficient tools. The nature of their job requires ability to multitask and give full focus at the same time. In addition, the quality of their work depends not only on their skill and knowledge, but also on successful collaboration with each other. For this reason, mistakes made in such context should not be considered solely as the sysadmins fault. Rather, it is the whole series of faults that resulted in an error, as Reason's (2000) Swiss cheese model explains.

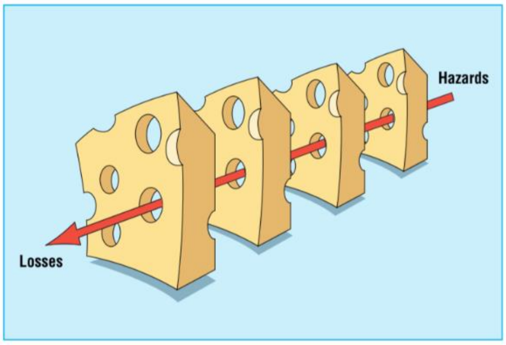


Figure 3. Swiss cheese model showing how multiple barriers need to fail to introduce major error (Reason, 2000, p 769).

The famous Swiss cheese model explains why errors occur by comparing each defence or a barrier that prevents errors, with a slice of Swiss cheese. In reality each barrier has weak spots – holes, which are constantly moving and changing place. A single hole in one of the barriers usually can be spotted and fixed without bigger issues. Only when a number of barriers fail to block the path of error, it results in a serious incident. The visualisation of the model is presented in the *Figure 3*.

There are two types of errors – active and latent, that contribute to the holes within the barriers (Reason, 2000). *Active errors* refer to human actions, such as slips and mistakes. *Slips* define accidents, when one action is intended, but another similar action is carried out instead, such as typing "lettr" instead of "letter" or hitting send instead of enter to go to new line, in the middle of chat message. Slips are unconscious errors, usually made when the person is familiar with the goal and does not pay full attention to the task (Laubheimer, 2015). In contrast, *mistake* defines the error, which happens when the goal chosen for the task is inappropriate, even if the steps to reach that goal were correct. For example, thinking that parameter *port* in configuration refers to communication from webserver to the authentication server settings, while in reality it is the other way around (Barret et al., 2004). There is no need to say that such situations would result in configuration error as the two situations require different parameter values. Mistakes are conscious errors that often arise from lack of understanding and incomplete or incorrect mental model formed about that system (Laubheimer, 2015). While slips are unavoidable, though they still can be prevented by designing interfaces that account for it. For example, showing suggestions when typing or choosing reasonable defaults. Mistakes, on the other hand, are harder to prevent. It requires the identification of what faulty models users form about the system and why. Then changes can be made to match the system closer to the user's expectation and to help the user form correct mental models of the system.

Another type of errors that contribute to the holes (weak points) in the cheese (the system) are *latent errors*. These errors refer to any decision made by builders, designers, and managers in creating and running the system, which, as Reason (2000) explains, either has potential to contribute to errors rising (e.g. tight deadlines, understaffing, poor tools), or adds up to the weak spots of the system (e.g. design, procedures). In the context of configuration management, it would take layers of insufficient tools, poor documentation, wrong interpretation of the system and either a slip or mistake to produce an error. However, in reality it depends on the system the sysadmin operates, tools available to them (e.g. configuration language and its framework), their skill and understanding of the system. Many times there are no design decisions that would not have any potential to turn into erroneous features of a system. It is not an easy task to design unambiguous, easily understood and straightforward configuration languages. It requires extensive knowledge about sysadmins, their expectation towards the system and mental models they have formed, in order to even begin to understand where the weak points in configuration languages lie.

2.2.4. Mental models

Mental models are conceptual models that represent the understanding of how something works. They are built from experience, observation (e.g. inferring devices and systems) and instruction (e.g. reading manuals) (Norman, 2013). Mental models include concepts, prototypes, categories, identities, stereotypes and are constantly updated, especially in cases when discrepancy between the outcome and the expectation is experienced. Different people can have different models of the same thing. People can also have multiple and even contradictory models for the same thing, in this case the environment dictates which model will be called out (World Bank, 2015).

In new situations we tend to rely on mental models for making decisions, they are like default assumptions we can rely on. Good models allow successful prediction of the consequences of actions. Without a mental model the user operates blindly following instructions and not being aware of what to do in the case of error (Norman, 2013). Since a big part of a system administrator job is troubleshooting, it is especially important to create tools which would allow users to form correct mental models about its behaviour.

Norman (2013) also noted that a good conceptual model does not need to involve all the inner workings of the system, it is enough to understand the relationship between the controls and the consequences. It is interesting to note, that people may even ignore all the evidence that does not agree with these default assumptions and instead fill in missing information based on their mental model rather than evidence (World Bank, 2015). Since we base our decisions on these models, the actual models system administrators form about the systems and tool they work with are significant.

Hebrec and Stiber (2001) were interested in uncovering the mental models of system administrators. They have conducted a survey questioning participants about what they know, what they don’t know and what they think about the system they are working with. 54 participants answered 8 questions about their perception of the system (generic vs. unusual), how much they felt they understood the system (0% vs. 100%), the main gap in their knowledge about the system (hardware vs. software), the source of their skill and knowledge (education vs. real world experience) and what actions are they most likely to take when something is not working (use of diagnostic tools vs. consult more experienced people vs. figure it out through experimentation). However, such questions do not reveal much specification of participant’s mental models. At most it provides information on whether they thought the system was difficult or not, and who they trusted for advice.

3. Concept development

In order to explore impact of ordering, we have first chosen the following features used in configuration languages: *inheritance, scope, referencing* and *ordering*, each of which are discussed in the following sections. Each of these features can have a few different underlying models and therefore differ in their implementation across languages. Since we discussed these features in a higher level, including variations of the implementations across Puppet, SmartFrog and L3, we refer to these features as *concepts*. The different models of the same concepts, later in this study, interchangeably are called *versions* and *interpretations* of these concepts.

Configuration tools are built according to some underlying *model*, which regulates the types of objects, relationships and operations that can be described. Different tools use different syntax to describe these models, however, that does not necessarily mean that the models also differ (Anderson, 2006). We studied features of Puppet, SmartFrog and L3 to find out whether they differ and how. These configuration languages are quite powerful and include a lot of features. Therefore, only a few could be discussed in this study. We aimed for features that appear in all three languages and seem to be particularly confusing, as it is more likely to be the weak spot of the language or languages. In addition, the purpose of a study like this is to discover the *holes* (weaknesses) in the *cheese* (configuration language) and suggest ways to fill those, before errors are exposed via incidents.

Qualitative data analysis was used to explore each of the languages, their syntax, behaviour, ordering rules and various features. It took a fair amount of time to reach the stage of the deeper understanding required, even with only a few of the features across three configuration languages. While SmartFrog and L3 appeared quite straightforward, Puppet, being the largest and most complex, took extra time to understand at the depth required for this study.

Referencing was chosen as a primary feature for further research, because we found it particularly confusing and adding a great deal of benefit for future languages if resolved. Referencing, as a concept, in the context of configuration management, stands for referring to previously declared attributes in order to reach their value. This feature is interrelated with inheritance, scoping and ordering, since each of these features has an impact on referencing, which is not always straightforward. We found that this group of features made up small enough but self-contained problem, suitable for this study. For this reason we took all of them into account and built our survey around it. All five features are described in detail in following sections.

3.1. Inheritance

Inheritance is a feature that allows to copy the structure, content and behaviour (or just either one) of one class to another avoiding duplication of the code. Inheritance can also be defined as allowing access from one class to another, rather than containing a copy of the original class content.

Configuration languages use *instance-based inheritance* (also called *prototype-based*), which differ from *class-based inheritance* used in many object-oriented programming languages. Using class-based inheritance the behaviour of the object is defined by the class, which is a blueprint of a particular type of object. In contrast, using instance-based inheritance, the object and its behaviour are first defined and then it is reused as a prototype for other objects.

Inheritance can be single, multiple or multilevel. *Single inheritance* means that there is only one base class (prototype), which was inherited, *multiple inheritance* means that the new class inherited from several original classes (prototypes). *Multilevel inheritance* means that the class inherited from another class, which itself was not original (prototype), but also inherited. Multilevel inheritance creates a hierarchy of classes.

In the simplest form, using our own pseudo code, single instance-based inheritance might look like this:

```
class-a {
   variable1 = value1,
   variable2 = variable2,
    }
class-b inherits class-a {
    variable1 = value4,
    variable3 = value3,
    }
```

Here, `class-b` inherits from `class-a`, meaning `class-a` is a prototype, also could be thought of as a parent to `class-b`. After inheritance, `class-b` has access to `variable1` and `variable2` values, as well as to `variable3`.

It depends on the model used by the configuration language, how inheritance is defined and whether order matters in it. Our pseudo code above can be interpreted in few different ways depending on desire, for example:

1. `class-a` contains `variable1` and `variable2`, while `class-b` contains `variable1`, `variable2` and `variable3`.
2. `class-a` contains `variable1`, `variable2` and `variable3`, while `class-b` contains `variable1` and `variable3`.
3. `class-a` contains `variable1` and `variable2`, `class-b` contains `variable1` and `variable3`.

Some implementations are more common across configuration languages than others. In addition, such interpretation as number 2 mentioned above was not found to be true in any of the three languages analysed. However, some novice users might still form such interpretation.

Inheritance can come in various shapes and forms, it is not always indicated using some key word, like *inherits* (Puppet) or *extends* (SmarFrog), it can also be noted just by the structure of the code itself. For example, in L3 inheritance as a feature is not present, instead, there is a feature called *composition*, which allows to compose partial configurations into one. Composition in L3 looks like this (Anderson, 2016a):

```
y: { b:1 } <+> { c:2}
```

Where b and c are different blocks being composed into one block y, the result of such composition is:

```
y: { b:1, c:2}
```

In Puppet, the classes refer to the collections of attributes, which then can inherit and override values from other classes. An example (Krum et al., 2013, p. 37):

```
node basenode {
   include sudo
   include mailx
}
node 'web.example.com'
   inherits basenode {
       include apache
       }
}
include ssh
```

The basenode and web.example.com both contain sudo and mailx classes, web.example.com node also contains apache. However, inheritance in Puppet it is highly discouraged. Puppet Style Guide notes (on docs.puppet.com):

> "Inheritance can be used within a module, but must not be used across module namespaces. Cross-module dependencies should be satisfied in a more portable way, such as with *include* statements or relationship declarations."

As mentioned in Puppet Style Guide, the *include* statement can be used to achieve essentially the same results as intended with inheritance and avoid code duplication. Using the *include* statement, class can be included in another class scope, allowing access to variables defined within that scope. Scopes are explained in more detail in the next section.

SmartFrog supports instance-based inheritance through prototyping, meaning that attribute values from the prototype can be inherited and also overridden with different values, if needed (Herry and Anderson, 2015). SmartFrog does not define types of components (classes), therefore any component can be a prototype for another (Goldsack, 2003). A simple example of prototype inheritance in SmartFrog (Goldsack, 2003, p. 3):

```
//webservertemplate.sf
webServerTemplate extends {
      sfProcessHost "localhost";
      port 80;
      useDB;
}
system extends {
      ws1 extends webServerTemplate {
      sfProcessHost "15.144.59.34"
}
```

In this example, `system`, `ws1` and `webServerTemplate` are all components with collections of attributes. Component `ws1` inherits from prototype component `webServerTemplate` using construct `extends` and the value of attribute `sfProcessHost` gets overridden.

Overriding is also supported in different ways across all three configuration languages. The model of overriding explains which value wins when the same variable was assigned different values, usually in different scopes.

In SmartFrog, the values within derived components override values in the prototype components. L3 uses *tags* to mark default values (`#default`), which should only take place if there is no other non-default value assigned, and final values (`#final`), which are preferred over all other values. In Puppet, most locally defined values within the immediate scope overrides less locally defined values from parent scopes.

These models of overriding are used only for the mentioned languages. However, there are also different interpretations novice users can make. Especially if lexical order is taken into account. One can interpret overriding in a way that global and parent variables always override local variables as it is declared first. Or, in contrast, one might think that the last declared value is correct no matter impact of other features. We discuss ordering issues in more detail in section 3.5.atter impact of other features. We discuss ordering issues in more detail in section 3.5.

3.2. Scope

Scope refers to the collection of variables or resources, grouped together and often enclosed in curly brackets { }, this is also called *class* in Puppet, *component* in SmartFrog and *block* in L3. Depending on the configuration language, there could be variety of scoping approaches used. For example, content within curly brackets can be allowed to be accessed from only inside of its own scope (the curly brackets), or also from outside of it.

Usually there are two scopes – local and global. *Local* variables refers to the variables declared inside of the block, component or a class, while *global* variables are all the variables defined outside of the component, class or block. Global variables are accessible from anywhere and local variables are accessible only from its own scope (hence the name). L3 and SmartFrog has these two scopes. Puppet, on the other hand, has stricter scoping rules.

As Krum et al. (2013) explained, there are four scopes always available in Puppet: top scope, node scope, parent scope and local scope. *Top scope* includes anything that is declared in the Puppet language file, called manifest (ending .pp). Anything written within the curly brackets of node definition belongs to the *node scope*. *Local scope* refers to a single class or defined type. *Parent scope* is created through inheritance, it is the local scope of original class and becomes parent scope to another class once inherited. Parent scopes can be assigned not just by inheritance, but also by declaration using `include` statement.

All the scopes in Puppet form parent-child relationships, where a child has an access to all of the parent scopes, but the parent does not have access to the child scopes. Therefore, a single class has access to its own local scope and also inherits from the parent, node and top scopes. In contrast, the node scope would inherit from top scope only.

Puppet identifies scope lookup rules, which determine when the local scope becomes a parent scope to another local scope (in docs.puppet.com). These rules are called *static* and *dynamic* scopes. The difference between them in Puppet is quite significant. In static scoping, parent scopes can only be assigned by the *inherits* keyword (and lambdas, but we are not considering it in this study). All the other scopes are just local scopes with no parents and have access to only the node and top scopes. Puppet documentation (in docs.puppet.com) states that static scopes are not dependent on evaluation order and their contents are determined by the class definition, as long as the class is declared inside the node scope, it has access to both top and node scopes. It also states that variable assignments are evaluation order dependant, meaning that variable cannot be resolved before it is assigned.

In contrast, *dynamic* scoping in Puppet means that parent scopes can be assigned not just by the *inherits* keyword, but also by declaration (*include*). Each scope has one parent, but can have unlimited number of grandparents, it also receives content from all of them with more local values overriding less local ones. The parent class is a base class and it is the first place where derived class has been declared. However, if the derived class is being declared, but its base class has not been yet declared, it is immediately declared in the current scope, which results in the base class being inserted between the current scope and the derived class. As one of the dynamic scope characteristics it is also mentioned that since classes can be declared many times using the *include* function, and that the contents of a given scope are evaluation order dependant (in docs.puppet.com).

Puppet used to use dynamic scoping for both variables and resource defaults, but not so long ago they have swapped to static scoping for variables. However, dynamic scoping is still used for resource defaults.

3.3. Referencing

Referencing is the action of calling a value of an attribute, defined elsewhere. Usually, when reference is used for a variable or resource which has not been declared yet, it is called a *future reference.*

In Puppet, references can be used for variables and resources. Static scope applies to variables, which means variables has to be declared before they are referenced to. Resource defaults are evaluated according to dynamic scope, therefore order issues do not apply. Using both static and dynamic scopes introduce inconsistency in Puppet reference feature. Therefore, system administrators would need to create several mental models to represent referencing within Puppet, and be able to recall the correct one each time when needed. It could be already confusing to recall referencing models for each different configuration language, having to store two of those for one language puts more cognitive load on the user and increases the risk of confusion resulting in mistakes.

Both variable and resource references can be accessed only in local, parent and grandparent scopes. For example, if we applied Puppet rules to our pseudo code:

```
Library {
    $type = student,

    Profile {
      account = $account,
      type = $type
    }

    $account = standard,
     }
}
```

The variable `Profile.type` would have a value of `student`, as it is declared before referencing in the parent scope. In contrast, the variable `account` would be `undefined`, since it is declared after referencing. The reference value, in Puppet, is first searched in the local scope, then in the parent scope and so on. In contrast, global variables are found at the top scope.

As already mentioned, depending on scope (static or dynamic), referencing will be handled differently. Wilkinson (2011) explained that if an *unknown* variable is being referred to under dynamic scope, the version of the variable used will be the one in scope at the time the function is called. In contrast, if an *unknown* variable is being referred to in static scope, the version used will be the one in scope within the enclosing code. Static scope allows variable binding to be checked and resolved before compiling, while in dynamic scoping it is unknown whether a compile will be successful and provide the desired outcome until it is compiled.

In L3, d*efault* references are interpreted first in the current block and then the next closest block is searched, while *absolute* references are interpreted in the top block

(Anderson, 2016a). In addition, reference to any resource can be used as a value. In such a situation, L3 evaluates both – static and dynamic versions of it and returns a composition of both values. Anderson (2016a) explains that in most cases only one of those values will be defined, therefore it will result in the only possible valid interpretation. For example (Anderson, 2016b, p. 2):

```
x: 1
a: {
      x: 2
      b: { c: { d:$.x } }
}
```

If *default* reference is being used, the variable `a.b.c.d.` value would result in `2`. If *absolute* reference is being used, `a.b.c.d.` would have value of `1`.

In SmartFrog, references can appear as a name of an attribute, on the left hand side, called *placement*, as a reference to the *prototype*, and as a *link* – an attribute value referring to another attribute, whose value is then copied, on the right hand side (Goldsack, 2003).

3.4. Ordering

It is often thought that in declarative languages order does not matter. However, we found that order does tend to matter in certain cases.

First of all, there are a few different orders in configuration languages, we separate two, which are lexical and evaluation order. *Lexical order* refers to the order in which configuration is written, while *evaluation order* refers to the order in which entities are evaluated. There are two ways entities can be evaluated: in an early manner (eager) and in a late manner (lazy). Early evaluation is used in static scope, while the late evaluation is used in dynamic scope.

3.4.1. Evaluation order

All three configuration languages support both early (static) and late (dynamic) evaluations.

L3 compiles both early and late evaluations and only provides tags to specify the evaluation order for the purpose of debugging and experimentation (Anderson, 2016a).

SmartFrog use tag *lazy* to indicate the need of late evaluation. Otherwise, references are resolved in the current context. Evaluation example in SmartFrog (Goldsack, 2003, p. 4):

```
// List of templates
# include “webservertemplate.sf”;
# include “dbtemplate.sf”;

system extends {
   commonPort “8080”;

   ws1 extends webServerTemplate {
       sfProcessHost
       port ATTRIB commonPort;
       useDB LAZY ATTRIB db;
   }
   ws2 extends webServerTemplate {
       sfProcessHost “13.144.59.64”;
       port ATTRIB PARENT:commonPort,
       type “backup”;
   }
   db extends dbTemplate {
       userTable:rows 6;
   }
}
```

In the example above, the variable `port` in `w1` is a reference to `commonPort` and will be replaced before deployment. In contrast, the variable `useDB` in `w1` with the tag *`LAZY`* will be resolved at runtime.

Puppet use static scope and, therefore, early evaluation for variables, but dynamic scope (late evaluation) for resources. In addition, Puppet has relationship dependency features which allow defining the resource evaluation order in relation to other resources, i.e. which of the two needs to be evaluated first, using statements like *require* and *notify*. Both statements work the same as the *include* statement, which means they both declare the resources. The only difference is that require and notify introduce some ordering (Larizza, 2014). The resource that is *required* by other resource, will be evaluated first, while the resource *notified* by other resource will be evaluated after the notifying resource is evaluated. Apart from this feature, the evaluation order is not in control of the Puppet user. Also, the relationship dependency feature alone implies that otherwise, order is not important.

3.4.2. Lexical order

As mentioned, lexical order refers to the way the code is written and structured. It is often perceived as unimportant, however, static scope as discussed in the section 3.2 proves that lexical order matters. In particular, it states that a variable has to be declared before it is referenced to. We found this to be important in Puppet.

Anderson and Herry (2015) noted that in SmartFrog it was very unclear, initially, whether the lexical order in the store was important. They discovered that in practice, language was often lexical order dependant. For example, in case of multiple inheritance, the order in which components are written will determine in which way they are composed. This does not quite go in line with the rest of the language. It is likely that novice system administrator would not be aware of such behaviour until running into errors. This, as in Puppet, adds more cognitive load on the user and requires memorizing such aspects of the language. In addition, it does not offer any cues to make recall easier. Anderson and Herry (2015) also found that the SmartFrog compiler supported forward references, while semantics did not.

In Puppet, lexical order is important for variables as static scope is used to evaluate them. Therefore, future referencing is not supported for variables, but is supported for resources.

L3 states that the lexical order within the block is not significant, which is in line with the popular view that order in declarative languages does not matter.

We were interested to see whether system administrators also found ordering to be insignificant in declarative languages and in our pseudo code. In addition, we were seeking to explore whether lexical order impacted their interpretation of the pseudo code presented.

4. Methodology

The purpose of the study is to research the expectation and intuitive judgement of system administrators, when presented with an unknown configuration language. We were interested to find out what mental models participants build regarding inheritance, scoping, referencing, lexical ordering and evaluation order. We then compared participants' preference to the models of these concepts existing in different configuration languages. To explore what mental models were most preferred across experienced, novice and potential (students and other interested) system administrators' a survey was designed, thoroughly tested and implemented using the *Bristol Online Survey tool*. The data collected was quantitatively analysed using *Microsoft Excel* and *Chi-square* test.

4.1. *Choosing the method*

Such exploratory research methods like *semi structured interview*, *case study*, *analysis of artefacts and experimental study* and *survey* were carefully considered. The *survey* method appeared to fit our goal, needs and capabilities the best.

Since the scope of the project has been narrowed down to a quite small and self-contained problem – *referencing*, the semi structured interview turned out to be irrelevant. Semi structured interviews are useful when collecting qualitative data. However, we were interested in collecting both qualitative and quantitative data. In addition, semi structured interviews are incredibly time consuming. In such a tight timeframe for the project it was decided to go for a more compatible method.

Originally, the main method was chosen to be case studies. It was expected to conduct few such studies with several participants and get them to solve presented examples in "thinking out loud" manner to be able to better understand their thought flow and how different details impact their judgement. However, since the course of the project changed its direction along the way, it was decided that case studies could not provide enough data for the amount of time it would take. This approach is especially suitable in the design stage of a product or programming language when there are some concrete goals or decisions to make. Such a method would point out the major flaws and could improve the usability greatly after only conducting it with very few participants.

The *survey* method has been chosen because it fit the requirements the best. An online survey can be reached in any place, internet connection being the only requirement. It can be distributed easily by sharing an URL and it can provide both quantitative and qualitative data without the need to conduct different studies. In addition, it takes the least amount of time to set up and time in which responses are being collected can be managed to fit tight deadlines.

4.2. Designing the survey

The *survey* method has been chosen as a best fit to the requirtements. An online survey can be reached in any place, internet connection being the only requirement. It can be distributed easily by sharing an URL and it can provide both quantitative and qualitative data without the need to conduct different studies. In addition, it takes the least amount of time to set up and time in which responses are being collected can be managed to fit tight deadlines.

The purpose of the study was to find out participant's intuitive judgements and expectations towards certain features in declarative languages. Such goal is quite abstract and there is no one uniform way to find out what mental models do users build in their head about things they interact with. If asked directly it is likely the answers will not be as informative or not even in line with actual actions. This could be impacted by misunderstanding, therefore more information lies in the observation.

Keeping in mind these constraints several survey designs were carefully considered before agreeing on a final approach. The survey had undergone 7 design – test – adapt cycles before it reached the desired stage and the deadline for developing it.

At the beginning of the survey development, presenting real code examples to the participants appeared to be a good choice. Such real code snippets were taken from Puppet and Smart Frog and participants were asked to match the outcome snippet to the configuration snippet that most likely produced the outcome. We have quickly learned that real configuration examples were too ambiguous for our needs: no concept could have been considered in isolation, because they are too closely related.

Changing configuration examples to be more isolated revealed another issue in the survey: the time it was taking to answer a single question was simply too long. The snippets were stripped to their essence of one value per answer only to discover that while focusing on the form, the purpose got forgotten. Trying to test intuitiveness in judgement while using real situational code was essentially the same as testing the knowledge of the configuration languages and their confidence in it. This is because different configuration languages support different versions of the same concept, therefore one should not expect that presenting the participant with particular language of the code will not influence him to use the version of the concept supported by that language.

To avoid this familiarity bias, it was decided to create our own pseudo code in a similar form to the analysed declarative languages, instead of real configuration examples. Such an approach reduced bias of the language used in the examples, and essentially shifted the purpose of the survey back to intuition and expectation, rather than a knowledge. We were more interested in what system administrators thought about the pseudo code presented to them, than what they knew about any particular language. Of course, every user will be biased by the configuration language they are familiar with, but pseudo code was expected not to introduce an additional bias.

4.3. Pilot test

Before launching the survey, the pilot test was carried out. A group of eight informatics student were asked to fill in the survey for ten minutes and then discuss the survey as a group. All the feedback collected during pilot test have been very valuable in the development of the shortest, most straightforward and least bias survey in the given time to serve the purpose of studying intuitive judgement in relation to configuration language features.

The test revealed the survey was taking significant amount of time to complete and initially contained too many confidence questions (after each regular question).

Participants shared that their level of confidence rose as they answered more and more questions, because they felt they were learning a new language and got more and more comfortable with it. Students revealed that certain statements or structure of the pseudo configuration made them bias towards static or dynamic scoping and that they were trying to choose approach that would make most situations work, purely because they would have liked to have more valid answers than errors in the survey. Such preference for validation made them choose dynamic scoping as a default mental model to rely on much more than static scoping. Otherwise, not all presented situations would have worked if a dynamic scoping approach was chosen.

It was also pointed out during the pilot study that the initial guess students had about the pseudo configuration, did not necessarily matched their preference towards it. All these comments were addressed in the final design cycle, many questions were removed, and pseudo code was adjusted to contain the least amount of statements and syntax symbols that could potentially make participants biased. In addition, comment boxes and a question about preferred ordering within the pseudo code was added.

Up until this stage the survey has been designed and tested on paper. In the pre-final stage the Bristol Online Survey Tool has been introduced and the survey has been moved to it. The survey then has been tested before launching on most popular browsers: *Safari*, *Chrome*, *Firefox* and *Internet Explorer* as well as on most popular operating systems: *Mac OS*, *Windows*, *Linux*, *iOS* and *Android* to make sure it appeared exactly the same on different screens and was not introducing an unexpected behaviour.

4.4. Implementation

The final design of the survey contained five pseudo configuration code examples, introducing various forms of previously discussed concepts: *inheritance*, *scoping*, *referencing*, *lexical ordering* and *evaluation order*. Participants were asked several questions about each example in a form of “Given the code above, what would you expect the value of variable *Post.length* to be?” where *Post.length* refers to a variable *length* within the block (can also be called component or a class) *Post*. All this type of questions were multi choice, but single answer. Answer options contained various values, almost each of which could be valid, depending on the interpretation (version) of the concept chosen. For example, if a participant chose to allow inheritance and overriding, they would choose the overridden value. In contrast, if a participant chose not to support inheritance and overriding, he might choose “undefined” as a value asked, or will choose an “error” as such feature is not supported. The difference between “undefined” and “error” also follows different interpretation of the concept. When the variable is “undefined”, it might mean that its value cannot be reached from within that class, but the variable has been declared elsewhere, while “error” could mean that such variable has not been declared anywhere and therefore the action of calling it results in error. To filter out confused participants, some answer options were chosen as invalid values, which do not map to any of our chosen interpretations of the configuration code. Otherwise, there were no incorrect or particularly correct answers. We were testing individual interpretations and preference, not knowledge.

In addition to multiple choice single answer questions, the survey contained open questions, asking about the mental models they chose to rely on during the task, their experience with declarative configuration languages, preference of ordering, most confusing aspects of configuration languages they have come across and other. Each page contained one pseudo code example, a few questions about it and a comment box, where participants were encouraged to leave their observations, questions and suggestions. The full survey can be found in Appendix A.

The survey was created using *Bristol Online Survey Tool*, because it has all the needed functionality and presentation. It also holds a contract with the University of Edinburgh. The survey was then distributed using the *snowballing method* – it was advertised to the targeted audience: experts in configuration language, authors of Smart Frog configuration language, people within Puppet community, few system administrator forums, as well as people who were more likely not to have experience in the field: students of University of Edinburgh (Informatics and other), members of Interaction Design Foundation and friends. The survey received interest from people in the Puppet community and therefore had been advertised within internal Puppet forums and personal blogs.

4.5. Data Analysis

The survey received vast interest and collected significant quantitative and qualitative data. Due to the scope of the project and the unusual format of the survey, we were not certain how successful the survey will be and what data analysis methods would be best to use. The methods, therefore, were chosen after carefully considering the data collected, once the survey was closed. *Descriptive analysis* method was used to introduce the amount and variety of the data.

We were interested to see whether there were any correlations between the models participants chose and their experience, education, configuration languages they know, confidence and similar. However, the survey appeared to be unsuitably designed to carry extensive statistical analysis. Quantitative data collected, even though in the form of frequency counts, was categorical, not numerical. For this reason, parametric statistics analysis used for correlation, such as *t-test* and *linear regression* could not be carried out. The data initially had one independent variable (demographics), but the number of dependant (each concept or versions of it) and independent variables (e.g. experience, confidence, age) can be manipulated, depending on what question is being asked.

Due to the data being categorical, and, therefore, any pair of independent and dependant variables also being categorical, Chi-square (χ^2) *test* was chosen. χ^2 is a statistical analysis method used for non-continuous variables such as frequencies and counts. It is most widely used to state the association between facts (Zibran).

In order to carry χ^2 analysis the data had to be processed first. Scale scores such as confidence, perception of easiness (i.e. *How easy was the question to answer?*) and level of skill were summed across the survey in such manner that each answer corresponding to 1 was multiplied by 1, and each answer corresponding to 5 was multiplied by 5, all the values then were added up and normalized to provide one digit score between 1 and 5 per participant, indicating confidence level, perception of easiness of the questions and skill level so then the frequency table could be created.

In addition, the answers from questions testing each of the concepts were mapped to corresponding versions of concepts also summed across, if more than one question was associated with the concept. Such observed frequency counts then were cross tabulated and expected frequencies were calculated. Expected frequencies were calculated according to the equation: $X^2 = \sum \frac{(O_i - E_i)^2}{E_i}$, where O_i stands for observed frequencies or counts and E_i – for expected counts, *i* runs through the cells in the table (Zibran). Then χ^2 test was applied. It takes both observed and expected frequencies and returns *p* value, which is probability that any correlation between entities is due to chance. Usually used threshold levels are α = 0.05 and α = 0.01. If the *p* value is smaller than the threshold value, the null hypothesis can be dismissed. Null hypothesis usually states that the two facts or entities are independent. In this study we used α level of 0.05. The data was processed using Microsoft Excel.

Due to time constraints on this project, qualitative data have not been processed. Some of participant’s comments were cited to illustrate and highlight certain issues. However, no statistical analysis has been carried out on the qualitative data, but further investigation is being considered out of the scope of this project.

5. Results

The survey was completed by 384 participants and an unexpected amount of quantitative and qualitative data was collected. Due to the time constraints we were not able to analyse the data extensively or even roughly address all of the aspects of it. Most of the qualitative data was put aside for the further analysis outside of the scope of this project. In this paper, we focused on few particular points in the data – what interpretations of concepts did participants prefer, how confidence, skill and experience correlates with their choices and what impact of ordering can be observed from participants answers and comments on the topic. These questions are addressed in more detail below.

5.1 Data description

343 (89.3%) participants were male and 18 (4.7%) were female, the other 23 (6%) preferred not to say. A third of participants admitted to have Bachelor's degree in IT or related field (see *Figure 4*). Master's and College degrees were also quite common across participants. Only a tenth of survey takers indicated not to have any formal education in IT or similar field.

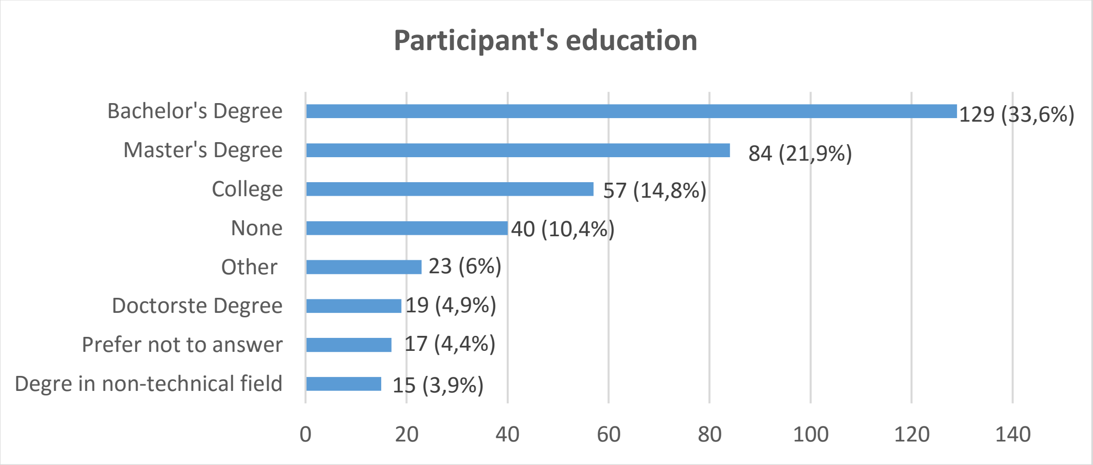


Figure 4. Participants' education.

This data does not support the earlier claim that most system administrators do not have a formal education in IT. However, since we cannot assure a random sampling of participants from the entire system administrator group, the sample could not be considered representative. Instead, it can reveal whether there are general issues that not only novice system administrators face in terms of configuration languages and encourage further investigation on this matter.

The youngest participant to take the survey was 15 years old and the oldest participant was 67 years old. 34 participants chose not to tell their age. For the rest of the participants, the mean of age was at 38.33, median at 37.5 and mode at 40 years old.

Almost third of participants indicated to have between 2 and 5 years of experience in configuration management and fifth said to have between 5 and 10 years of experience (see *Figure 5*).

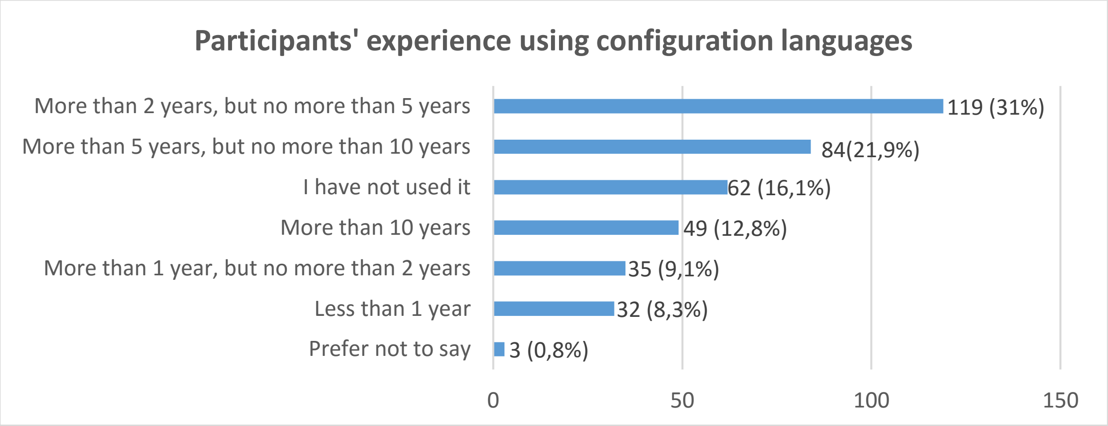


Figure 5. Participant's experience in years using configuration languages.

In addition to the education and working experience, we have also asked participants some questions to determine their skill levels. Participants had to identify how much they agree to given statements in a scale from 1 to 5, where 5 is *Strongly agree* and 1 is *Strongly disagree*. The statements included such actions like writing a program that sorts numbers in a file or using configuration management tools to install software on multiple computers (full list questions can be found in Appendix A). The responses then have been summed for each participant. 5 points were given for each *Strongly Agree* and 1 point for each *Strongly Disagree*. Therefore, the maximum score a participant could get was 25, while the smallest was 0, in case option *I don't know* was chosen. As it can be seen from the *Figure 6* most participants strongly agreed to given statements and were confident assessing their skill.

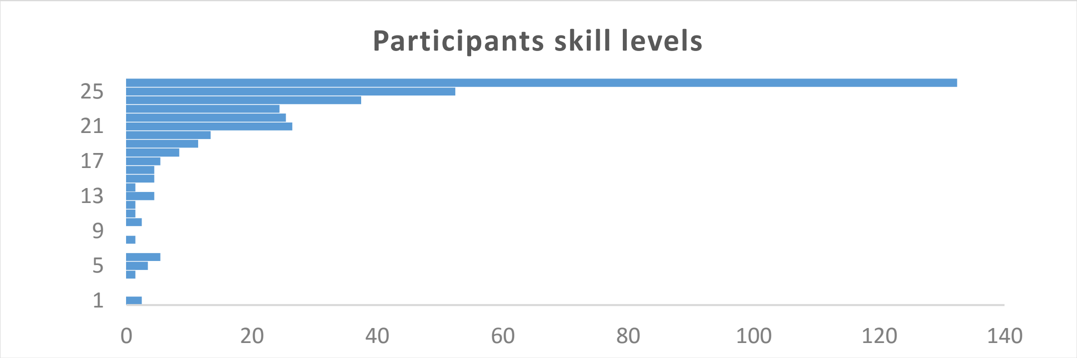


Figure 6. Participants skill levels summed across five questions.

These skill scores later were normalized to fit the scale from 1 to 5, due to carrying analysis with Chi-square test.

5.2 Confidence levels

Five times during the survey participants were asked to indicate their confidence level in a scale from 1 to 5, 1 being *Not at all confident* and 5 being *Very confident*. This number corresponds with the number of examples they were presented with. As shown in *Figure 6* participants' confidence differed a lot.

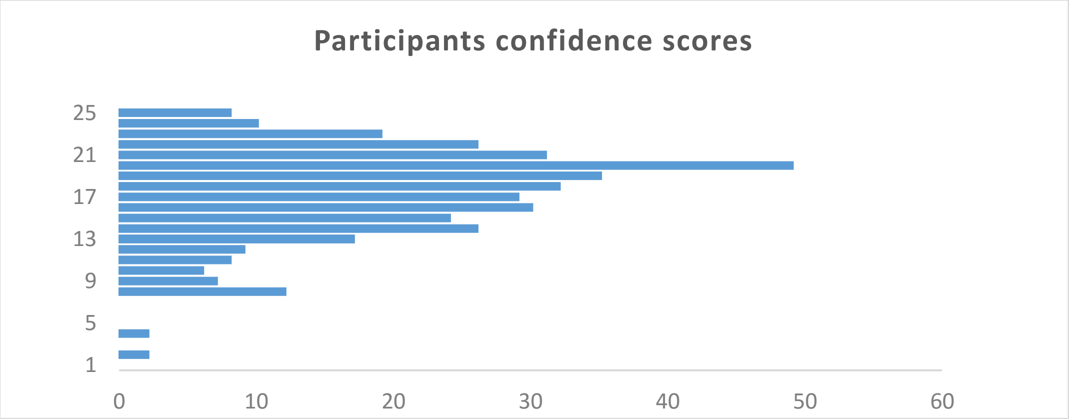


Figure 7. Participants' confidence scores across the survey.

In the pilot study, discussed in section 4.3 some participants said they were getting more confident as the number of answered questions increased. We were interested to see whether this trend would be visible in the data collected. It appeared that the opposite was rather true – overall confidence declined towards the end of the survey.

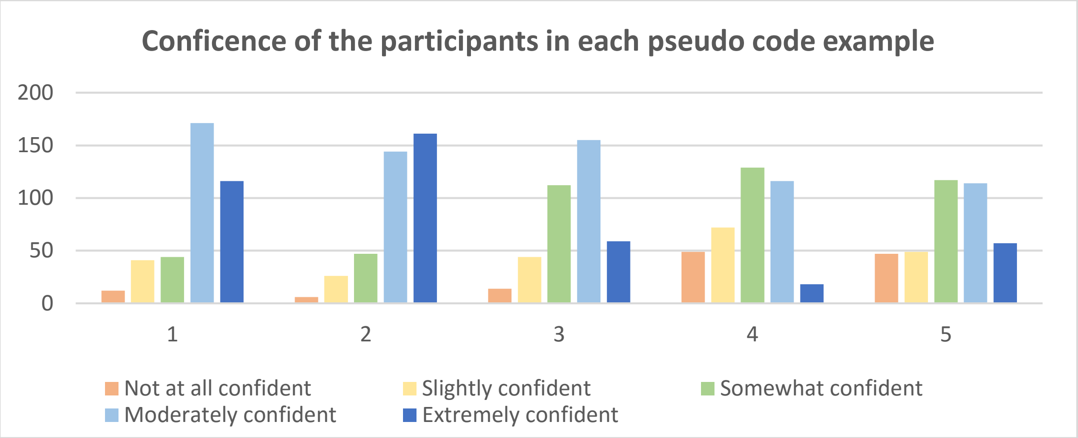


Figure 8. Confidence scores of all participants after answering questions about each of the five pseudo code examples.

The results go in line with the perception of how easy were the questions about each example (see *Figure 9*).

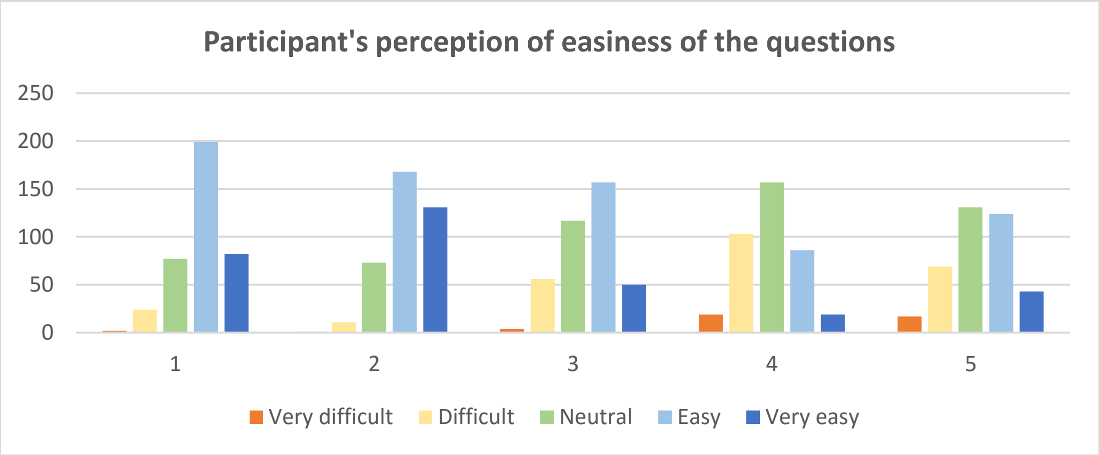


Figure 9. Change in participants' perception of easiness through the survey.

The overall score of perception of easiness was also counted in the same manner as overall confidence score – summing across. As shown in the *Figure 10*, participants had very different experiences in terms of difficulty.

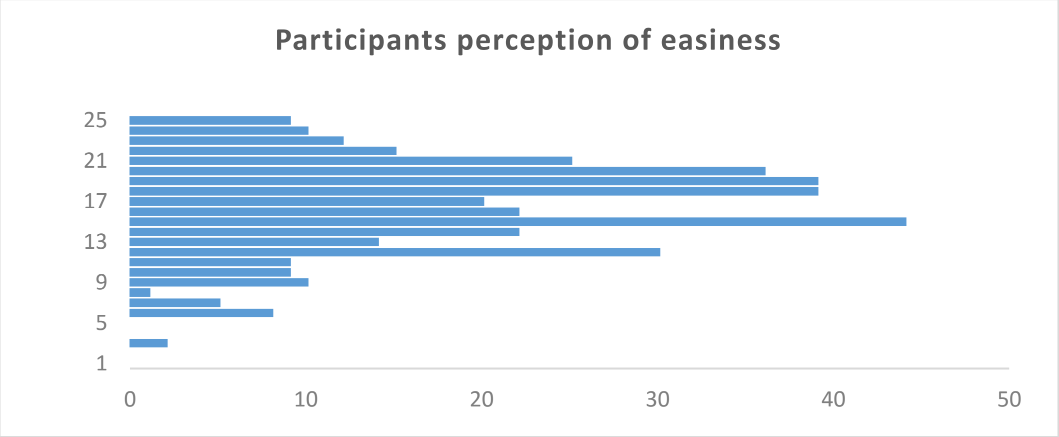


Figure 10. Overall perception of easiness by participants, where 25 indicates the survey was very easy.

A t-test revealed positive correlation between having Bachelor's or Master's degree and higher confidence (p < 0.01), in comparison to having no degree. However, no other comparison with demographic data was statistically significant.

5.3 Mental models of concepts

Results showed that two thirds of participants chose to interpret *inheritance* in such a way that inheriting class received the content from the prototype class, and were consistent thorough the survey (see *Table 1).* A third of participants were only partially consistent, meaning they have chosen some answers that do not support the standard interpretation of inheritance. In particular, a number of participants seemed to support reversed inheritance, expecting the prototype class to receive the content of derived class. For example, presented with the example 1 of:

[example 1]

```
Message  {
      colour = red,
      length  = 231,
      font = Ariel,
}

Post inherit Message {
      comments = 15,
}
```

Asked, what they expect the value of variable ***Message.comments*** to be, participants chose all kinds of answers (see *Figure 11*).

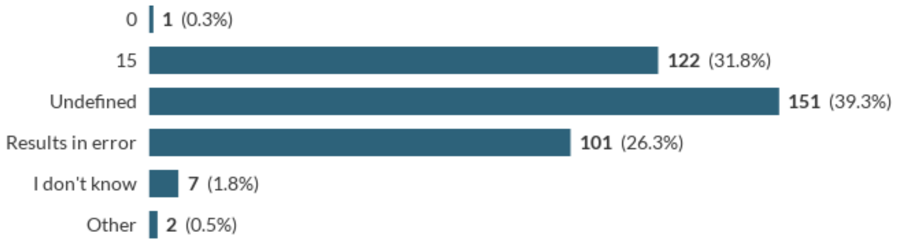


Figure 11. The distribution of participant's answers to question 1 of the survey.

Similar behaviour was also captured in one more question of the same kind, given code:

[example 2]

```
Mountain-bike inherit Bicycle {
        tyre = 26,
        kickstand = true,
}

Bicycle {
        wheels = 700,
        tyre = 28,
        kickstand = false,
        brakes = true,
}
```

And asked about the value of the variable, participants again chose a range of answers (see *Figure 12*).

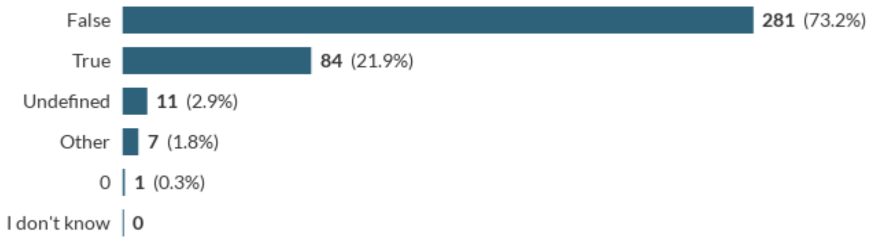


Figure 12. The distribution of participants answers to question 8 of the survey.

In addition, when similar situation was presented again, almost all participants chose the same answer. Given this code:

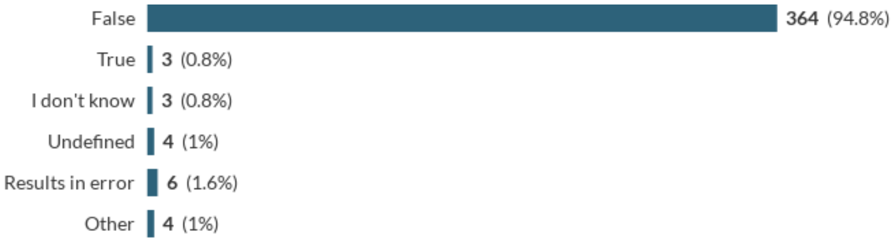
[example 3]

```
Box {
    show-balance = true,
}
```

```
Player {
        tracks = 573,
        genres = 11,
        show-balance = false,
}

Music-box inherit Box inherit Player {
        genres = 9,
}
```

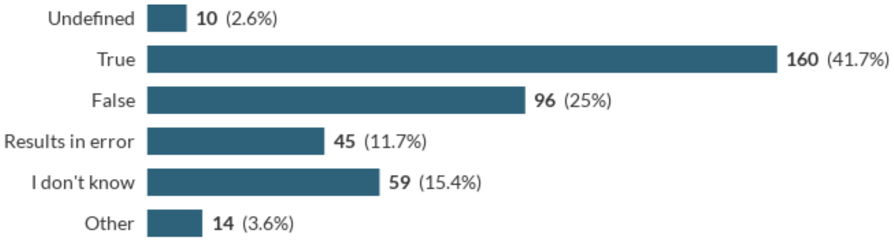
And asked about value of the variable ***Player.show-balance***, suddenly participants agreed with each other (see *Figure 13*).



There seem to be significant correlation between consistency towards inheritance model and confidence (p = 0.000728).

Multilevel inheritance was interpreted in more than two ways by the participants. The most popular approach was to follow the pattern of regular inheritance. In this way, if *A inherits* from *B* and *B inherits* from *C*, the *B* values would override the values inherited from *C*, and *A* would override values inherited by *B*. However, it was followed by less than half of the participants. The second most common approach was to assume that the last prototype in the chain (most to the left) would override the values on the right. In this case, if *A inherits* from *B* and *B inherits* from *C*, the *A* would receive values from *C* only, and those might override the values within *A* as well.

When asked what is the value of variable ***music-box.show-balance*** in the code [example 3] presented previously, participants did not agree:



There were no significant correlations between the choice of multiple inheritance model and gender, education or working experience or confidence.

Participants were mostly torn between two models of forward referencing. A little over half participants assumed that forward referencing is only allowed for classes, but not variables, similarly like in Puppet. We call this version as *partially supported* or *Puppet like*. Another 40% of participants decided to allow forward referencing for both classes and variables, it is named as *fully supported* in the *Table 1*. No statistical significance was found between chosen referencing interpretation and gender, education or experience in working with configuration languages.

Almost 95% of participants interpreted the scope as dynamic for classes. Meaning, they found order not important and also supported forward referencing for the classes. For variable scope, both static and dynamic interpretations were chosen equally often. There were no significant correlation between chosen scope and gender, working experience or education of participants.

Ordering within pseudo code was interpreted both as important and non-important by participants. Almost half of the participants were not consistent in the way they interpreted order. Meaning that they have sometimes treated it as significant, and in other times as insignificant. The number of participants being inconsistent in their ordering interpretation is very close to the number of participants who preferred the model of static scoping, as opposite to dynamic.

Table 1. Frequency scores of concept interpretations (rows) and gender, education and working experience (columns).

		Gender			Education							Work experience			
	Total	Female	Male	n/a	None	College	Bachelor's	Master's	Doctorate	Non technical & other	n/a	1>	1to5	5<	n/a
Inheritance	**383**														
Consistent standard	234	8	211	15	21	30	88	53	10	25	7	45	100	87	2
Partially consistent	126	8	110	7	15	24	37	24	7	10	8	40	45	41	0
Inconsistent/Confused	23	2	21	0	3	3	11	7	2	3	1	8	9	5	1
Multilevel inheritance	**371**														
Follows regular inheritance	160	7	142	11	20	29	54	34	6	4	13	35	72	51	2
Last one mentioned wins	97	7	86	4	7	18	28	26	7	4	7	26	41	30	0
Confused	59	3	52	4	8	6	18	10	3	3	11	16	21	21	1
Do not support	55	1	50	4	5	2	23	11	3	3	8	16	18	21	0
Ordering	**383**														
Inconsistent/Confused	176	7	157	13	22	25	58	44	5	17	9	43	75	58	1
Order matters	131	6	120	4	11	16	43	28	11	15	6	33	46	51	1
Order does not matter	76	5	66	5	7	16	28	12	3	6	2	18	33	24	1
Variable scope	**371**														
Static scope variables	171	8	155	8	20	22	57	39	12	5	16	39	74	57	1
Dynamic scope variable	169	10	147	12	16	28	60	35	5	7	18	44	69	55	1
Confused	31	0	28	3	4	6	7	7	2	1	4	10	8	13	0
Class scope	380														
Static scope resource	17	1	16	0	2	2	3	5	2	0	3	7	3	6	1
Dynamic scope resource	363	17	323	23	38	54	125	79	16	14	37	87	149	125	2
Forward reference	**382**														
Partially/Puppet like	214	9	190	15	18	34	74	46	10	23	9	54	81	78	
Support fully	163	10	142	10	13	26	48	40	9	18	8	41	64	58	
Do not support	5	0	5	0	0	0	2	2	1	0	0	2	2	1	

5.4 Confusion over ordering

In addition to all the quantitative questions, we asked participants to comment on whether they think the order was important in the pseudo code presented. It was not specified what order we asked about. A variety of opinions were received on the topic. Many participants preferred interpretations in which order did not matter, according to declarative approach:

> "I interpreted code order as irrelevant."
>
> "I interpreted it as order independent."
>
> "Order per context unimportant."
>
> "About code order, I tried to interpret that did not matter."
>
> "Order doesn`t matter."
>
> "It looks declarative so I interpreted order as not mattering, like in other declarative languages."
>
> "I didn't spot any cases where ordering would have been significant to interpretation. "
>
> "Order does not matter, inheriter can override inherited values, inheriting same value multiple times is an error, $variables are lexically scoped"

In contrast, a few defined order as important, since they were not following a declarative approach:

> "I interpreted them like in most procedural / object-oriented languages. Not as declarative languages."
>
> "Strict ordering, as listed."
>
> "'Inherit' keyword was most probably a trait of object oriented programming and the order was important."
>
> "I started with text order, so things are defined in sequence."
>
> "I decided code order mattered arbitrarily, perhaps on a cue from seeing line numbers."

Several participants stated explicitly what order they considered important and therefore took into account. These range across several features. For example, evaluation order:

> "Unordered compile-time, ordered run-time."
>
> "I assumed that the code was executed in the order it appeared, as if it was interpreted in a single pass (rather than compiled)."

> “I interpreted code order in a way that a computer would read through it and execute it in order and with indentations denoting layers.”
>
> “I initially assumed code order didn't matter re scope, visibility. It should only matter in terms of execution sequence (imho!)”

Lexical order of variable and within the class appeared to be important to a few participants:

> “I worked on the assumption that order within a class was significant, but that order of class declaration was not.”
>
> “Code order matters in assignments.”
>
> “Order of properties should not matter. Order of variables should.”
>
> “Roughly java style -- order within functions matters, but not class/type definitions. Diamond inheritance problems should probably result in error over attempting to use ordering as a solution”

Some comments provide insight in how participants rank various features in terms of their importance in the code. A few participants have discussed such relationship between ordering and inheritance:

> “Code order doesn't need to matter, but styling wise inheritance is more obvious if they're in order.”
>
> “I only considered order important where two objects were inherited from. Not important where objects were defined.”
>
> “Line by line order seemed a bit more flexible. But on a line for inheritance was harder to judge.”
>
> “I though order was not important but I took inheritance into account.”
>
> “From code order it was hard to tell how to interpret. Much easier when there were explicit inherits from statements.”
>
> “Subclasses override superclasses, order matters less than inheritance, strict scoping.”
>
> “In one example there was a resource/class that was defined with inheriting another resource/class. In this case I made the example of violating order matters as the class above was referencing the later class directly.”

Scoping appeared to be another concern for participants in the context of ordering:

> “Scope > order but order looked important.”
>
> “I chose to only take scope into account, as ordering is much less trivial”
>
> “Since there is scope, as indicated wiht {}, i would expect order to be irrelevant”
>
> “Well, order shouldn't matter, but it confuses things. Back to the question: It's all about scoping. Values at an innner scope, or in a derived class, override values at the outer scope or base class.”

And since scoping could not be thought of without referencing, the latter was also mentioned by participants:

> “I have a strong preference for languages in which order does not matter, while scope matters (more local definition takes precedence). Thus, I tried to interpret the snippets as such. As for multiple definitions, anything that introduced ambiguities as how to interpret it should result in an error.”

> “It seems logical to assume that if a variable is defined, then subsequently changed then it would take on the new value - that is whole point of a variable.”

> “If code was out of order, just pretended it wasn't, but for multiple variable declaration, I chose the last time if it was still in scope, or error if it was out of scope.”

However, quite a few comments indicated that people were not even sure what models they used to evaluate situations, whether order mattered or not. Or revealed to have used different models according to the code presented:

> “It's just confusing.”

> “It's complicated.”

> “I choose to not interpret them if they did not seem obvious.”

> “I think order should matter, but it wasn't clear from this code if the language cared or not.”

> “If the language contained definite statements as to how it was ordered, then I used them. If not, I'm not prepared to guess.”

> “Languages that looked more like interpreted languages I already know, I assumed might depend on order. Others that appeared declarative or looked like compiled languages I decided order might not matter.”

A few non-feature specific comments revealed that order does tend to cause confusion, especially when most expected not to be important:

> “I usually thing that for a configuration language, order must not be relevant, so I think that all the time. But I have being burnt in past to do not assume too much just by looking at the code, there are always some tricks and gotchas.”

> “I believe code order is more of a restriction, than trying to provide order, but this is what are we used too. Variables could be lexically scoped using blocks.”

> “Code order often matters, but class order may not. Referencing classes for inheritance before they are defined may be understood by a compiler and re-ordered; accessing class variables vs instance variables may not be detected properly. This varies wildly by language, but many modern languages have a tendency to minimize the impact of ordering, but is not a guarantee regardless of language age. Additionally, inheritance,

access of undefined parameters, and access of undefined variables also vary. In fact, in some languages, there is no traditional "definition" for variables, they are simply instantiated on the first reference. How the language handles these decisions can have cascading impacts on my interpretation of the language usage.”

6. Discussion

In the need to keep IT services up and running at all times and in the context of expensive, yet unavoidable, configuration errors, the goal of finding ways to close the gap between inexperienced system administrators and the tools designed for experienced system administrators is crucial in order to stop human error.

In this study we suggested approaching human mistakes from a Human Factors perspective, by exploring what is it that system administrators fail to understand about the system, why, and how could it be designed to allow easier adoption of correct mental models. We have studied several features of three declarative configuration languages (Puppet, SmartForg and L3) and built a survey to collect information about the participants' interpretations of unfamiliar pseudo code, aiming to uncover features of the languages that cause the most confusion and lead to mistakes. By exploring these erroneous parts of the languages we hoped to help improve existing configuration languages and create new and better configuration tools.

Results revealed that even though most participants felt quite confident, the overall levels of confidence declined thorough the survey. There could be various reasons behind this, but one way to look at it would be considering the design of the survey. The two first examples were very similar in terms of layout of the code and questions asked, the confidence levels for the first and second examples were also extremely high in comparison to the rest of the survey. Still, the second example questions received the highest confidence scores, possibly because at the time of answering those, participants would have seen similar pseudo code already twice. It could have confirmed the initial model they decided to adopt and led to an expectation that following examples would align in the same manner. However, the following examples introduced different concepts and might have challenged the adopted model. Therefore, the confidence levels started to decline.

It appears that over half of the participants felt familiar with the *inheritance* concept and were consistent in its interpretation throughout the survey. However, up to a third of the participants occasionally selected such answers that supported the reverse model of inheritance. Meaning, that due to inheritance, the prototype class receives the content of derived class. We hypothesised that such unexpected and inconsistent judgement could be explained in one of the two following ways. It is either the result of rushing, lack of focus and misreading the questions, thinking that the class mentioned is a derived class, because of similar questions before and after in the survey, or confusion caused by lexical ordering of the code, especially since ordering is the only thing that differed across those three situations, which all received different participant's interpretations (presented in section 5.3).

Over 50 participants marked *I don't know* when questioned about multilevel inheritance. The remaining participants split into three groups. There is a group that prefers multilevel inheritance unavailable as a feature, as it might reduce confusion. The other two groups supported the multilevel inheritance, but in different ways. One group of participants applied standard inheritance rules, while the other interpreted in such way that the last defined (most left) prototype takes over the classes on the right. However, we cannot be entirely sure that such model was really chosen by all the participants, as it could be that some other rules appeared more important to the participants. For example ordering. Participants might not care much about order of inheritance, but care about the order of declaration instead, and therefore the value marked is based on that interpretation.

Dynamic scoping for classes was supported consistently by nearly all of the participants. Considering that most participants were experienced and familiar with Puppet, it seemed fair that this concept did not introduce any confusion. Likely for this reason, *future referencing* was mostly supported in a partial manner, used in Puppet – future referencing allowed for classes, but not for variables, due to different scope rules. Less than half participants chose to interpret future reference as fully allowed for both variables and classes. Similar distribution could be seen in the interpretation of variable scope. Almost an identical number of people chose both static and dynamic scope for variables. Such equal division is an interesting indicator that none of the two is significantly worse than the other, or could possibly also indicate misunderstanding of what a declarative language should look like rather than what it looks like.

Participants' interpretation of the ordering concept appeared to be especially uneven. It is likely that participants supporting partial ordering come from Puppet background, since the mixture of static and dynamic scopes introduce inconsistency in ordering across. However, the other two groups of people chose to either take ordering as absolutely important or not. Such division could be purely based on practice and declarative versus imperative configuration tools usually used by the participants. Such division in the interpretations of the participants and also some of their comments, presented in the section 5.4 suggest, that the code itself is not clear on whether order is important or not. A few participants have also indicated that they were changing their interpretation of each example code (the survey had 5 in total) depending on what previously used language it reminded them of. This indicates that since many system administrators know many different languages, they also need to carry several mental models in their head and be able to correctly recall the one needed with often little environment support. From the comments in the section 5.4 we have learned that our pseudo code was insufficient to provide enough cues and decide whether certain features were important and how one should expect them to work.

6.1 Conclusion

System administrators are often required to know multiple languages and tools and be able to use them accordingly. This means that they are required to form several mental models for often the same features, but corresponding to different languages. They are then required to recall the needed model correctly at all times, based on limited cues from the environment, such as curly brackets, nesting or certain keywords.

Quicker and more successful recall of models could be accommodated by keeping the consistency of configuration language syntax in line with other languages sharing the same features and functionality and providing distinctive and uncommon cues for features that are specific to the language.

Such distinction between *common* (known) and *specific* (new) would allow the user to rely on his previous experience when working with known features and would help build new models that are not confused with other similar features across the languages that might have similar names, but different functionality and vice versa.

Order is often thought of as unimportant in declarative configuration languages. More specifically, lexical order seems unimportant because the configuration defines the state, not the process. In addition, evaluation order seems unimportant because the declarative framework will sort it out for the user without his knowledge. But these views are considering ordering rather in a higher level as a single feature out of any context. Therefore, it is not surprising that it is perceived as unimportant.

In contrast, the results from our study shows that once you put the ordering feature in the context of configuration language, it has high potential to result in a lot of confusion.

Even in the Puppet, order matters when it is thought of in the context of static variable scope. Order also matters when constructing an inheritance chain for multilevel inheritance. You would not want to get all the overriding wrong. It matters in the context of referencing, as the user might not always be aware when the reference becomes future reference. Participants who took our survey struggled despite their education, working experience, confidence and skill, because they did not know whether the order was important and how to find it out. It leads them to adopt some mental models that often were inconsistent. The same way novices are not aware of internal specifics apart from what is in front of them and are encourage to build incorrect mental models without guidance. Therefore the cues indicating the functionality of the language and consistency across would allow mental models to be built quicker, easier and more successfully.

6.2 Limitations

The survey provided a lot of quantitative and qualitative data and suggested some insights into the usability of configuration languages and ordering impact. This project was quite abstract and open ended, and since there were no guidelines on how to conduct research on mental models people form about systems, the study has a number of limitations. However, a lot of lessons have been learned in the process.

First of all, all the results were interpreted in a way that we thought was fair, but subjectiveness in such case cannot be avoided. The pseudo code created for the survey, the mapping between answers and concepts, the interpretation of results, it is all based on human judgement, therefore can be considered subjective. In addition, the pseudo code was created gathering inspiration from the studied configuration languages and therefore syntax and structure could have primed the participants to choose one interpretation over another.

The subjective interpretation of results could have been avoided if the survey had been designed in a suitable way for quantitative analysis. In addition, even though we tried to avoid as much bias in our survey and pseudo code, it is impossible to provide completely unbiased survey. One day after launching we have received comments from participants indicating that since Puppet was used as an example in several statements across the survey, it appeared as we were priming participants to interpret code in a Puppet manner.

The survey undergo several design cycles, but mistakes were still not avoided. Several notations in the questions were not clear enough to indicate what was asked for and this impacted participants' judgement. The last example turned out to be faulty (even though it still produced interesting results), but was only noticed after the survey was closed.

In addition, the participants lacked feedback to be able to interpret how such pseudo would code work. Providing participants with static representation of the system and asking them to form dynamic model of it is just not fair or realistic. However, it was only understood at the end of the project when there was no time to change things.

6.3 Future work

One of the main limitations was that participants were asked to evaluate the code without actually being able to click *compile* button. If done over new this project would have been focused on creating a small debugging game to accommodate for interactive environment to test the same mental models formed around same concepts. Such approach might also appear more attractive to inexperienced people than the survey.

However, since this study was exploratory and open ended, there are many things that could still be looked at in more detail. For example, similar surveys could be promoted in different than Puppet community circles to be able to compare preferred mental models of same concepts. Also, different aspects of languages could be addressed in similar manner, considering the limitations we have found and documented.

Bibliography:

Anderson, P. & Herry, H. (2015). A Formal Semantics for the SmartFrog Configuration Language. *Journal of Network and System Management*. 1-37. DOI: http://dx.doi.org/10.1007/s10922-015-9351-y

Anderson, P. (2006). *System configuration*. SAGE short topics in system administration, USENIX Association.

Anderson, P. (2016a). *Composition and Reference in Declarative Configurations*. Draft of 2015/10/26, 16:11. University of Edinburgh.

Anderson, P. (2016b). *The L3 Configuration Language*. Draft on 2016/06/14, 11:23. University of Edinburgh

Attariyan, M., Flinn, J. (2010). Automatic Configuration Troubleshooting with dynamic information analysis. *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 237-250.

Barrett, R., Kandogan, E., Maglio, P. P., Haber, E. M., Takayama, L. A., Prabaker, M. (2004) Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. Proceedings of the 2004 ACM conference on Computer supported cooperative work, 388-395. DOI: 10.1145/1031607.1031672

Border, C., & Begnum, K. (2014). Educating System Administrators. *USENIX; login:* 39, 5 (October 2014), 36-39.

Bozman, J. S., Perry, R. (2010). *The business value of large-scale server consolidation*. IDC Whitepaper. Available at: http://www-05.ibm.com/innovation/cz/leadership/pdf/POL03073USEN.pdf (Accessed: 9 August, 2016).

Brodkin, J. (2012). Why Gmail Went Down: Google Misconfigured Load Balancing Servers. Available at: http://arstechnica.com/information-technology/2012/12/why-gmail-went-down-google-misconfigured-chromes-sync-server (Accessed: 11 April 2016).

CircleID.com. (2009). *Misconfiguration Brings Down Entire .SE Domain in Sweden*. Available at: http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden/ (Accessed: 11 April 2016).

Collard, J., Gupta, N., Shambaugh, R., Weiss, A. & Guha A. 2015. On Static Verification of Puppet System Configurations. Available at: http://arxiv.org/pdf/1509.05100v3.pdf (Accessed: 4 August 2016).

Dapeng, J. (2007). Personal Firewall Usability – A Survey. Seminar of Network Security. Available at: http://www.tml.tkk.fi/Publications/C/25/papers/Jiao_final.pdf (Accessed: 11 April 2016)

Dong, Z., Ghanavati, M., Andrzejak, A. (2013). Automated Diagnosis of Software Misconfigurations Based on Static Analysis. *Proceedings of the 2013 International Conference on Software Engineering*, 312-321.

Ford., B. (2014). Puppet's Declarative Language: Modelling Instead of Scripting. Available online: https://puppet.com/blog/puppet%E2%80%99s-declarative-language-modeling-instead-of-scripting (Accessed: 15 August 2016).

Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., Toft, P. (2003). *SmartFrog: Configuration and Automatic Ignition of Distributed Applications*. A technical overview paper from the 2003 HP Openview University Association conference. Available at: http://www.hpl.hp.com/research/smartfrog/papers/SmartFrog_Overview_HPOVA03.May.pdf (Accessed: 9 August, 2016).

Gray, J. (1986). Why do computers stop and what can be done about it? *Symposium on Reliability in Distributed Software and Database Systems.*

Hrebec, D., G., Stiber, M. (2001) A Survey of System Administrator Mental Models and Situation Awareness. In *Proceedings of the ACM SIGCPR Conference*. (pp. 166-172).

Johnson, R. (2010). More details on today’s outage. Available at: http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919 (Accessed: 11 April 2016).

Kandogan, E., Maglio, P., Haber, E., & Bailey, J. (2012). Taming Information Technology: Lessons from Studies of System Administrators. Oxford University Press, USA.

Kanies, L. (2012). Why Puppet has its own configuration language. Puppet.com. Available at: https://puppet.com/blog/why-puppet-has-its-own-configuration-language (Accessed: 6 August, 2016).

Keller, L., Pupadhyaya, P., Candea, G. (2008). ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. Proceedings of the IEEE International conference on Dependable Systems and Networks with FTCS and DCC, 2008. DOI: 10.1109/DSN.2008.4630084

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., and Wiedenbeck, S. (2011). The state of the art in end-user software engineering. ACM Comput. Surv. 43, 3, Article 21 (April 2011), 44 pages. DOI: http://doi.acm.org/10.1145/1922649.192265

Krum, S., Van Heveling, W., Kero, B., Turnbull, J., McCune, J. (2013). Pro Puppet. 2nd Edition. Apress Berkely, CA, USA. ISBN:1430260408 9781430260400

Larizza, G. (2014). On Dependencies and Order. Available online: http://garylarizza.com/blog/2014/10/19/on-dependencies-and-order/ (Accessed: 10 August).

Laubheimer, P. (2015). Preventing User Errors: Avoiding Unconscious Slips. Available online: https://www.nngroup.com/articles/slips/ (Accessed: 10 August, 2016).

Liang, L. Y. (2013). Linkedin.com inaccessible on Thursday because of server misconfiguration. Available at:

http://www.straitstimes.com/singapore/linkedincom-inaccessible-on-thursday-because-of-server-misconfiguration (Accessed: 11 April 2016).

Lindberg, H. (2014). Getting Your Puppet Ducks in a Row. Available at: http://puppet-on-the-edge.blogspot.co.uk/2014/04/getting-your-puppet-ducks-in-row.html (Accessed: 11 April 2016).

Mahajan R., Wetherall D., Anderson T. (2002). Understanding BGP Misconfiguration. *SIGCOMM'02*, August 19-23.

McNamara, P. (2009). Opinion: Missing dot drops Sweden off the Internet. Available at: http://www.computerworld.com/article/2529287/networking/opinion--missing-dot-drops-sweden-off-the-internet.html (Accessed: 11 April 2016)

Mette, A., Hass, J. (2002). *Configuration Management Principles and Practice*. Published Dec 30, 2002 by Addison-Wesley Professional. Part of the Agile Software Development Series.

Mikoliunaite, A. (2016). Usability of System Configuration Languages: Errors Caused by Ordering. Informatics Research Proposal. University of Edinburgh

Miller, R. (2012). Microsoft: Misconfigured Network Device Caused Azure Outage. Available at: http://www.datacenterknowledge.com/archives/2012/07/28/microsoft-misconfigured-network-device-caused-azure-outage/ (Accessed: 11 April 2016).

Nagaraja, K., Oliveira F., Bianchini, R., Martin, R. P., Nguyen, T. D. (2004). Understanding and Dealing with Operator Mistakes in Internet Services. *In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04) Volume 6*, Pages 5-5.

Nielsen, J., Molish, R., (1990). Heuristic evaluation of user interfaces. Proceedings in CHI '90 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 249-256.

Norman, D. (2013). *The Design of Everyday Things*. Revised and Expanded edition. Basic Books, A Member of the Perseus Books Group.

Norman, D. A. (1983). Design Rules Based on Analyses of Human Error. *Communications of the ACM.*

Oppenheimer D., Ganapathi D., and Patterson D. A. (2003). Why Do Internet Services Fail, and What Can Be Done About It? *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, March 2003.

Patterson, D. A. (2002). A Simple Way to Estimate the Cost of Downtime. *Proceedings of LISA '02, 185-188.*

Reason, J. (2000). Human errors: models and management. BMJ 320(7237): 768-70. DOI: http://dx.doi.org/10.1136/bmj.320.7237.768

Ritter, E. F., Baxter, G. D., Churchill, E. F. (2014). Foundations for Designing User-Centered Systems: What System Designers Need to Know about People. Springer-Verlag London. DOI: 10.1007/978-1-4471-5134-0

Sverdlik, Y. (2014). Microsoft Says Config. Change Caused Azure Outage. Available at: http://www.datacenterknowledge.com/archives/2014/11/20/microsoft-says-config-change-caused-azure-outage/ (Accessed: 11 April 2016).

The Aws Team. (2011). *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*. Available at: https://aws.amazon.com/message/65648/ (Accessed: 11 April 2016).

The Aws Team. (2012). *Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region*. Available at: https://aws.amazon.com/message/680587/ (Accessed: 11 April 2016).

The Google Apps Team. (2013). *Google Apps Incident Report*. Available at: http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/us/appsstatus/ir/ej73a82sddnv7fb.pdf (Accessed: 11 April 2016).

Traugott, S., & Brown, L. (2002). Why Order Matters: Turing Equivalence in Automated System Administration. *USENIX; LISA '02*. 99-120.

Velasquez, N. F., Weisband, S., and Durcikova, A. (2008). Designing Tools for System Administrators: An Empirical Test of the Integrated User Satisfaction Mode. *22nd Large Installation System Administration Conference (LISA '08)*

Weilli, F., Cheney, J., Anderson, P. (2016) *An operational semantics for a fragment of the Puppet Configuration Language*. Technical report, August 2016, available at http://arxiv.org/abs/1608.04999

Wilkinson, D. (2011). Lexical scope and function closures. Available online: https://darrenjw.wordpress.com/2011/11/23/lexical-scope-and-function-closures-in-r/ (Accessed: 10 August).

Wong, T. (2008). On the Usability of Firewall Configuration. SOUPS. July, 23-25. DOI: 10.1.1.222.7037

Wool, A. (2004). A Quantitative Study of Firewall Configuration Errors. Computer. Vol. 37 (6). 62-67. DOI: 10.1109/MC.2004.2

Wool, A. (2009). Firewall Configuration Errors Revisited. DOI: 10.1.1.157.1615

World Bank. (2015). Chapter 3: Thinking with Mental Models. World Development Report 2015: Mind, Society, and Behavior. Washington, DC: World Bank. DOI: 10.1596/978-1-4648-0342-0.

Xu, T. & Zhou, Y. (2015). Systems approaches to tackling configuration errors: A survey. ACM Comput. Surv. 47, 4, Article 70 (July 2015), 41 pages. DOI: http://dx.doi.org/10.1145/2791577

Xu, T., Pandey, V., & Klemmer, S. (2016). An HCI View of Configuration Problems. Project Report for Human-Computer Interaction course. University of California. Available at: http://arxiv.org/pdf/1601.01747.pdf (Accessed: 11 April 2016).

Xu, T., Zhang, J., Huang, P., Zheng J., Sheng, T., Yuan, D., Zhou, Y., & Pasupathy, S. (2013). Do Not Blame Users for Misconfigurations. SOSP’13, Nov. 3–6, DOI: 10.1145/2517349.2522727

Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., Pasupathy, S. (2011). An empirical study on configuration errors in commercial and open source systems. SOSP '11. 159-172. DOI: 10.1145/2043556.2043572

York, D. (2015). HBO NOW DNSSEC Misconfiguration Makes Site Unavailable From Comcast Networks (Fixed Now). Available at:

http://www.internetsociety.org/deploy360/blog/2015/03/hbo-now-dnssec-misconfiguration-makes-site-unavailable-from-comcast-networks-fixed-now/ (Accessed: 11 April 2016).

Zibran, M., F. CHI-Squared Test of Independence. Department of Computer Science University of Calgary, Alberta, Canada.

Appendix A