2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Does Your Configuration Code Smell?

Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis
Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{tushar,mfg,dds}@aueb.gr

ABSTRACT

Infrastructure as Code (IaC) is the practice of specifying computing system configurations through code, and managing them through traditional software engineering methods. The wide adoption of configuration management and increasing size and complexity of the associated code, prompt for assessing, maintaining, and improving the configuration code's quality. In this context, traditional software engineering knowledge and best practices associated with code quality management can be leveraged to assess and manage configuration code quality. We propose a catalog of 13 implementation and 11 design configuration smells, where each smell violates recommended best practices for configuration code. We analyzed 4,621 Puppet repositories containing 8.9 million lines of code and detected the cataloged implementation and design configuration smells. Our analysis reveals that the design configuration smells show 9% higher average co-occurrence among themselves than the implementation configuration smells. We also observed that configuration smells belonging to a smell category tend to co-occur with configuration smells belonging to another smell category when correlation is computed by volume of identified smells. Finally, design configuration smell density shows negative correlation whereas implementation configuration smell density exhibits no correlation with the size of a configuration management system.

CCS Concepts

•**Software and its engineering** → **Specification languages; Software maintenance tools;** *Software libraries and repositories;* Software design engineering;

Keywords

Infrastructure as Code, Code quality, Configuration smells, Technical debt, Maintainability

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901761

1. INTRODUCTION

Infrastructure as Code (IaC) [13] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through traditional software engineering methods. For example, a server farm that contains numerous nodes with different hardware configurations and different software package requirements can be specified using configuration management languages such as Puppet [39], Chef [37], CFEngine [4], or Ansible [1] and deployed automatically without human intervention. Such automated yet custom configured deployment is not only faster than the manual process but is also reliable and repeatable.

Apart from automating an infrastructure deployment, the IaC paradigm brings the infrastructure, the code and the tools and services used to manage the infrastructure, in the purview of a software system. Therefore, IaC practices treat configuration code similar to the production code and apply traditional software engineering practices such as reviewing, testing, and versioning on configuration code as well.

A lot of work has been done to write maintainable code [8, 21] and achieve high design quality [36] in traditional software engineering. Similar to production code, configuration code may also become unmaintainable if the changes to configuration code are made without diligence and care. In a recent study, Jiang et at. [14] argued that configuration code must be treated as production code due to the characteristics and maintenance needs of the configuration code. Therefore, traditional code and design quality practices must be adopted to write and maintain high quality configuration code.

In this context, we planned a preliminary quality analysis of configuration code where we focused on the maintainability aspect of the configuration code quality. We pose the following questions to achieve the above stated goal:

1. What is the distribution of maintainability smells in configuration code? Which smells are commonly found and which ones are rarely found?

2. What is the relationship between the occurrence of design configuration smells and implementation configuration smells?

3. Is the principle of coexistence applicable to smells in configuration projects?

4. Does smell density depend on the size of a configuration project?

To answer these research questions, we compiled a catalog of 24 configuration smells. We used the existing Puppet-Lint [27] tool and extended its rule-base to identify implementation configuration smells. We developed a tool namely *Puppeteer* to identify cataloged design configuration smells. We extracted 4,621 repositories containing Puppet code from GitHub and applied the tools on all the repositories. Using the collected data, we provide the empirical answers to each of the research questions listed above.

Some key observations from the study are the following.

- Configuration smells belonging to a smell category tend to co-occur with configuration smells belonging to another smell category when correlation is computed by volume of identified smells.
- Design configuration smells show 9% higher average co-occurrence among themselves than the implementation configuration smells.
- Design configuration smell density shows negative correlation whereas implementation configuration smell density exhibits no correlation with the size of a configuration management system.

Studying code quality of configuration management systems can help us understand the characteristics of configuration code. This study can benefit the researchers by exposing them to a method for investigating system configuration practices by mining repositories. At the same time, practitioners can identify configuration smells using the tools employed in this study and adopt best practices to write maintainable configuration code.

The rest of the paper is structured as follows: Section 2 provides an overview of the study with a goal and corresponding research questions. Section 3 describes a catalog containing implementation and design configuration smells. Section 4 details the developed tool, mining Puppet repositories from GitHub and applying the tool on the repositories. Section 5 provides detailed answers to the research questions. We discuss related work in Section 6.

2. OVERVIEW OF THE STUDY

We planned a preliminary study to analyze the existing configuration code and evaluate the associated code quality to examine the existing practices towards keeping configuration code maintainable. Although, there are many attributes and sub-attributes of code quality (such as reliability and portability), the focus of this study is on the maintainability attribute of configuration code quality.

We formulated the following research questions towards the quality analysis goal of configuration code.

RQ1. *What is the distribution of maintainability smells in configuration code?* We investigate the distribution of configuration smells to find out whether there exists a set of implementation and design configuration smells that occur more frequently with respect to another set of configuration smells.

RQ2. *What is the relationship between the occurrence of design configuration smells and implementation configuration smells?* We study the instances of design configuration smells and implementation configuration smells to discover the degree of co-occurrence between the two categories of configuration smells.

RQ3. *Is the principle of coexistence applicable to smells in configuration projects?* In traditional software engineering, it is said that patterns (and smells) co-exist as "No pattern is an island" [3] i.e. if we find one, it is very likely that we will find many more around it [3, 36]. We investigate the intra-category co-occurrence of a smell with other smells to find out whether the folklore is true in the context of configuration smells. Furthermore, whether all the smells in each of the categories follow the principle with a same degree.

RQ4. *Does smell density depend on the size of the configuration project?* Smell density is a normalized metric that represents the average number of smells identified per thousand lines of code. We investigate the relationship between the size of a configuration project and associated smell density for both the smell categories to find out how the smell density changes as the size of the configuration project increases.

We present a theoretical model of configuration smells to study the research questions listed above. The model contains two categories of configuration smells namely *implementation configuration smells* and *design configuration smells* with a brief description of each smell.

To detect the majority of implementation configuration smells, we used Puppet-Lint [27]. Due to the lack of an existing tool that can detect design configuration smells, we developed a tool namely *Puppeteer* for detecting the cataloged design configuration smells.

We identified repositories containing Puppet code and downloaded them from GitHub [10]. We downloaded 4,621 repositories containing 142,662 Puppet files and 8.9 million lines of code and analyzed them with the help of Puppet-Lint and Puppeteer. We grouped the information collected based on the data required to answer the research questions and deduce our observations.

We assume that the reader of this paper has a fair idea about Puppet and syntax of its programming language. If it is not the case, we encourage the reader to familiar himself/herself with configuration management [33] through Puppet [26].

3. THEORETICAL MODEL: CONFIGURATION SMELLS

We define configuration smells as follows:

Configuration smells are the characteristics of a configuration program or script that violate the recommended best practices and potentially affect the program's quality in a negative way.

In traditional software engineering practices, bad smells are classified as implementation (or code) smells [8], design smells [36], and architectural smells [9] based on the granularity of abstraction where the smell arises and affects. Similarly, configuration smells can also be classified as implementation configuration smells, design configuration smells, documentation configuration smells, and so on. In this paper, our focus is on two major categories of configuration smells namely implementation configuration smells and design configuration smells.

We assigned a three letter acronym starting from *I* for each implementation configuration smell and from *D* for each design configuration smell. We use these acronyms in the later sections of the paper.

3.1 Implementation Configuration Smells

Implementation configuration smells are quality issues such as naming convention, style, formatting, and indentation in configuration code. After studying available resources, such as the Puppet style guide [35] and rules implemented by Puppet-Lint, we prepared a list of recommended best practices. We grouped the best practices based on their similarity and arrived at a corresponding implementation configuration smell when a best practice is violated. Table 1 lists the implementation configuration smells and corresponding set of best practices.

Here, we present a list of implementation configuration smells with a brief description. Figure 1 shows an annotated Puppet example with all the cataloged implementation configuration smells.

Missing Default Case (IMD) A default case is missing in a *case* or *selector* statement.

Inconsistent Naming Convention (INC) The used naming convention deviates from the recommended naming convention.

Complex Expression (ICE) A program contains a difficult to understand complex expression.

Duplicate Entity (IDE) Duplicate hash keys or duplicate parameters present in the configuration code.

Misplaced Attribute (IMA) Attribute placement within a resource or a class has not followed a recommended order (for example, mandatory attributes should be specified before the optional attributes).

Improper Alignment (IIA) The code is not properly aligned (such as all the arrows in a resource declaration) or tabulation characters are used.

Invalid Property Value (IPV) An invalid value of a property or attribute is used (such as a file mode specified using 3-digit octal value rather than 4-digit).

Incomplete Tasks (IIT) The code has "FIXME" and "TODO" tags indicating incomplete tasks.

Deprecated Statement Usage (IDS) The configuration code uses one of the deprecated statements (such as "import").

Improper Quote Usage (IQU) Single and double quotes are not used properly. For example, boolean values should not be quoted and variable names should not be used in single quoted strings.

Long Statement (ILS) The code contains long statements (that typically do not fit in a screen).

Incomplete Conditional (IIC) An "if..elsif" construct used without a terminating "else" clause.

Unguarded Variable (IUV) A variable is not enclosed in braces when being interpolated in a string.

3.2 Design Configuration Smells

Design configuration smells reveal quality issues in the module design or structure of a configuration project. Various available sources, such as the Puppet style guide [35], blog entries [16, 17], and videos of technical talks [18] highlight the best practices to be followed for configuration code. We obtained a list of commonly occurring design configuration smells from the violation of these best practices at design-level. We assigned relevant names (many a times inspired by the traditional names of smells) to the smells and documented their forms representing variations of the smells. Here, we present design configuration smells with a brief description.

Multifaceted Abstraction (DMF) Each abstraction (e.g. a resource, class, 'define', or module) should be designed to specify the properties of a single piece of software. In other words, each abstraction should follow single responsibility principle [20]. An abstraction suffers from *multifaceted abstraction* when the elements of the abstraction are not cohesive.

The smell may occur in the following two forms:

- a resource (file, package, or service) declaration specifies attributes of more than one physical resources, or
- all the language elements declared in a class, 'define', or a module are not cohesive.

Unnecessary Abstraction (DUA) A class, 'define', or module must contain declarations or statements specifying the properties of a desired system. An empty class, 'define', or module shows the presence of *unnecessary abstraction* smell and thus must be removed.

Imperative Abstraction (DIA) Puppet is declarative in nature. The presence of imperative statements (such as "exec") defies the purpose of the language. An abstraction containing numerous imperative statements suffers from *imperative abstraction* smell.

Missing Abstraction (DMA) Resource declarations and statements are easy to use and reuse when they are encapsulated in an abstraction such as a class or 'define'. A module suffers from the *missing abstraction* smell when resources and language elements are declared and used without encapsulating them in an abstraction.

Insufficient Modularization (DIM) An abstraction suffers from this smell when it is large or complex and thus can be modularized further. This smell arises in following forms:

- if a file contains a declaration of more than one class or 'define', or
- if the size of a class declaration is large crossing a certain threshold, or
- the complexity of a class or 'define' is high.

Duplicate Block (DDB) A duplicate block of statements more than a threshold indicates that probably a suitable abstraction definition is missing. Thus a module containing such a duplicate block suffers from *duplicate block* smell.

Table 1: Mapping Between Implementation Configuration Smells and Corresponding Best Practices

Smells	Best practices
Missing default case	Case and Selector statements should have a default case
Inconsistent naming convention	The names of variables, classes and defines should not contain a dash
Complex expression	Expressions should not be too complex
Duplicate entity	Duplicated hash keys and parameters should be removed
Misplaced attribute	• "ensure" attribute should be the first attribute specified • The required parameters for a class or 'define' should be listed before optional parameters • Right-to-left chaining arrows should not be used
Improper alignment	• Properly align arrows (arrows are not all placed one space ahead of the longest attribute) • Tabulation characters should not be used
Invalid property value	• "ensure" property of file resource should be valid • File mode should be represented by a valid 4-digit octal value (rather than 3) or symbolically • The path of "puppet:///" URL should start with "modules/"
Incomplete tasks	"FIXME" and "TODO" tags should be handled
Deprecated statement usage	Deprecated node inheritance and "import" statement should not be used
Improper quote usage	• Booleans should not be quoted • Variables should not be used in single quoted strings • Unquoted node names should not be used • Resource titles should be quoted • Literal boolean values should not be used in comparison expressions
Long statement	Lines should not be too long
Incomplete conditional	"if ... elsif" constructs shall be terminated with an "else" clause
Unguarded variable	Variables should be enclosed in braces when being interpolated in a string

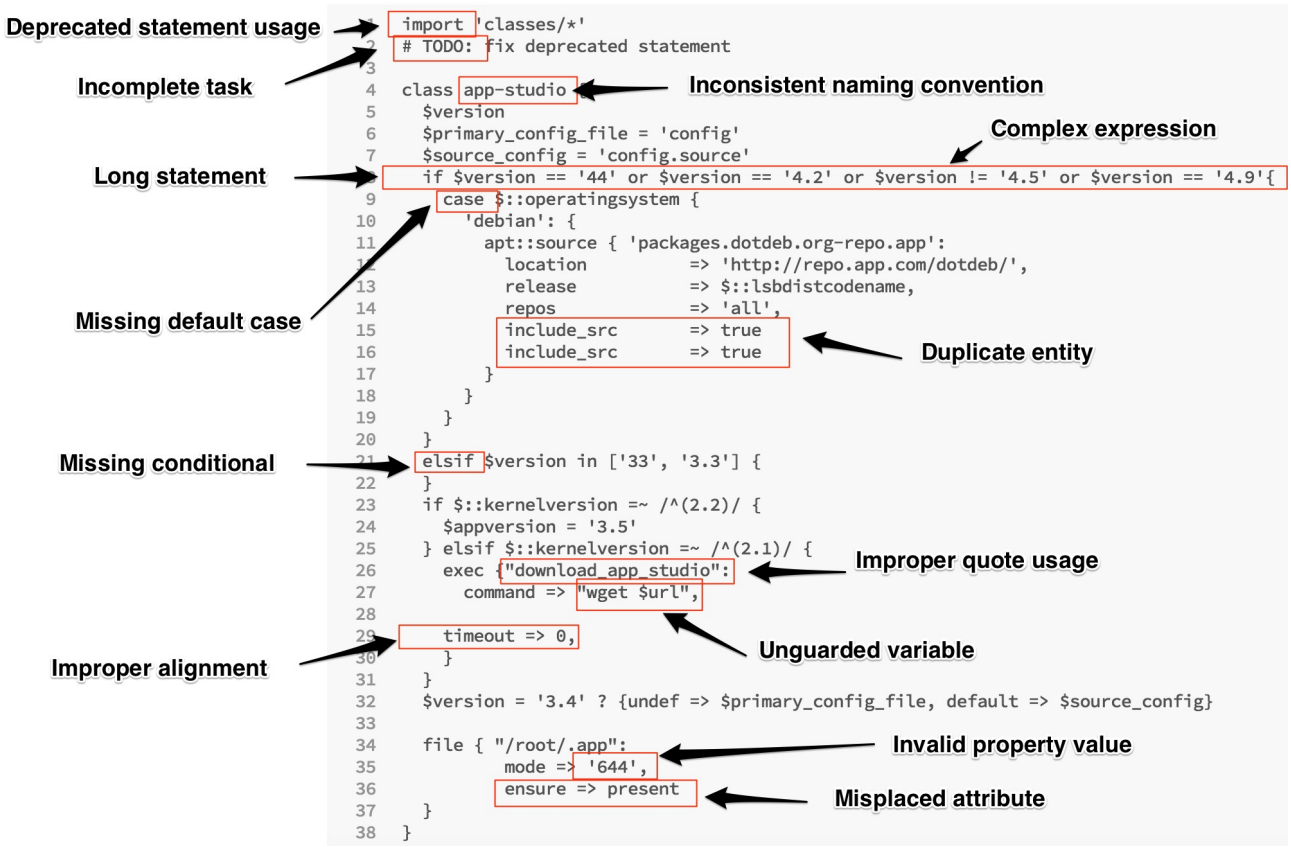

Figure 1: An annotated Puppet example with all the cataloged implementation configuration smells

Broken Hierarchy (DBH) The use of inheritance must be limited to the same module. The smell occurs when, the inheritance is used across namespaces where inheritance is not natural ("is-a" relationship is not followed).

Unstructured Module (DUM) Each module in a configuration repository must have a well-defined and consistent module structure. A recommended structure for a module is the following.

Module name

```
manifests
files
templates
lib
facts.d
examples
spec
```

An ad-hoc structure of a repository suffers from *unstructured module* smell that impacts understandability and predictability of the repository.

Dense Structure (DDS) This smell arises when a configuration code repository has excessive and dense dependencies without any particular structure.

Deficient Encapsulation (DDE) This smell arises when a node definition or ENC (External Node Classifier) declares a set of global variables to be picked up by the included classes in the definition.

Weakened Modularity (DWM) Each module must strive for high cohesion and low coupling. This smell arises when a module exhibits high coupling and low cohesion.

4. MINING GITHUB REPOSITORIES

We followed the procedure given below to select and download the repositories.

1. We employed GHTorrent [11, 12] to select GitHub repositories to download.

2. There are various options to choose from to select the subject systems such as number of commits and committers, stars, and number of relevant files in the repository. Each of the options (or their combinations) present different tradeoffs. For instance, there are only 838 Puppet repositories that have five or more stars. We wanted to analyze larger number of repositories to increase the generalizability of the observations. Reducing the number of stars as a selection criterion would have resulted in more number of repositories; however, the significance of the criterion would have reduced. A high number of commits in a repository shows continuous evolution and thus we chose number of commits as the selection criterion. We chose to download all the repositories where the number of commits was more than or equal to 40. The above criterion provided us a list of 5,387 Puppet repositories to download.

3. We were able to download 4,621 repositories except some private repositories.

Table 2: Characteristics of the Downloaded Repositories

Attributes	Values
Repositories	4,621
Puppet files	142,662
Class declarations	132,323
Define declarations	39,263
File resources	117,286
Package resources	49,841
Service resources	18,737
Exec declarations	43,468
Lines of code (Puppet only)	8,948,611

Table 2 summarizes the characteristics of the downloaded repositories. We observed, by random sampling, that the downloaded repositories were either standalone Puppet-only repositories or system repositories (where production code as well as configuration code has been put together into a repository).

4.1 Analyzing Puppet Repositories

We analyzed the downloaded repositories to detect implementation and design configuration smells. We used PuppetLint tool to detect majority of implementation configuration smells. We executed the tool on all the repositories and stored the generated output. In addition to use PuppetLint, we wrote our custom rules to detect the implementation configuration smells that the tool was not detecting (for instance, *complex expression* and *incomplete conditional*). We then aggregated the number of individual implementation smells that occurred in each repository using the generated output and our mapping of best practices to the implementation smells (see Table 1).

We developed a tool, *Puppeteer*,[1] to detect design configuration smells listed in Section 3.2. We discuss detection strategies of all the smells detected by Puppeteer in the next subsection.

The data generated by the tools mentioned above for both the smell categories for all the analyzed repositories can be found here.[2]

4.2 Design Configuration Smells - Detection Strategies

This section discusses detection strategies that Puppeteer uses to identify design configuration smells.

Multifaceted Abstraction The detection strategy for the two forms of the smell is as follows.

1. We compute a metric, "Physical resources defined per resource declaration", for each declared resource. We report the smell when the metric value is more than one.

2. We compute lack of cohesion for the configuration abstractions to detect the second form of the smell. Traditional software engineering uses the LCOM (Lack of Cohesion Of Methods) [5] metric to compute lack of cohesion for an abstraction. The same metric cannot be used for configuration

[1]https://github.com/tushartushar/Puppeteer
[2]https://github.com/tushartushar/configSmellData

code due to its different structure and characteristics. We use the following algorithm to compute LCOM in a configuration code abstraction.

(a) Consider each declared element (such as file, package, service resources and exec statements) as a node in a graph. Initially, the graph contains the disconnected components (DC) equal to the number of elements.

(b) Identify the parameters of the abstraction, used variables, and literals (such as file name). Call them as data members collectively.

(c) For each data member, repeat the following: identify the components that uses the data member. Merge the identified components in a single component.

(d) Compute LCOM:

$$LCOM = \begin{cases} \frac{1}{|DC|} & \text{if } |DC| > 0 \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

Note that we compute LCOM for each class, 'define', and file. Therefore, it is quite possible that the tool reports more than one instance of this smell in a single Puppet file.

Unnecessary Abstraction We compute a metric namely "Size of the abstraction body". A zero value of the metric shows that the abstraction doesn't contain any declarations and thus suffers from *unnecessary abstraction* smell.

Imperative Abstraction We compute a metric namely "Total 'exec' declarations" in a given abstraction. The tool reports the *imperative abstraction* smell when the abstraction has more than two 'exec' declarations and ratio of the 'exec' declarations against all the elements in the abstraction is more than 20%.

Missing Abstraction We identify total number of configuration elements except classes or defines that are not encapsulated in a class or 'define'. A module suffers from the smell if there are more than two such elements in the module.

Insufficient Modularization The detection strategy for the three forms of the smell is as follows.

1. We count the number of classes and defines declared in a Puppet file. We report the smell if a file defines more than one class and 'define'.
2. We count the number of lines in an abstraction. If a class or 'define' contains more than 40 lines of code, it suffers from the smell.
3. We compute maximum nesting depth for an abstraction. An abstraction with maximum nesting depth more than three suffers from this smell.

Duplicate Block We use the PMD-CPD [6] tool to identify code clones. A module suffers from this smell when a code clone of larger than 150 tokens gets identified in the module.

Broken Hierarchy For all the class definitions, we identify the inherited class (if any). If the inherited class is defined in any other module, the class suffers from "broken hierarchy" smell.

Unstructured Module The detection strategy for the three forms of the smell is as follows.

1. We search for a folder named "manifests" in the root folder of the repository. If the total number of Puppet files in the folder is more than five while there is no folder containing the string "modules", the smell gets detected.
2. We find a folder containing the string "modules" and treat all the sub-folders as separate modules. Each module must have a folder named "manifests". Absence of the folder shows the presence of the smell.
3. In each module, we count the unexpected files and folders. Expected files and folders are: "manifests", "files", "templates", "lib", "tests", "spec", "readme", "license", and "metadata". A module with more than three such unexpected files or folders suffers from the smell.

Dense Structure We prepare a graph for each repository to detect the smell. Each module is treated as a node and any reference from the module to another module is treated as an edge. We, then compute average degree of the graph.

$$AvgDegree(G) = \frac{2 \times |E|}{|V|} \quad (2)$$

where $|E|$ and $|V|$ are number of edges and nodes respectively. A graph more than 0.5 average degree suffers from *Dense structure* smell.

Deficient Encapsulation We count the number of global variables declared for each node declaration, followed by at least one include statement. If a node declaration has one or more such global variables, the module suffers from *deficient encapsulation* smell.

Weakened Modularity We compute *modularity ratio* [2] for each module as follows:

$$ModularityRatio(A) = \frac{Cohesion(A)}{Coupling(A)} \quad (3)$$

where, $Cohesion(A)$ refers to the number of intra-module references and $Coupling(A)$ refers to the number of inter-module references from module A. We report the smell if the ratio is less than one.

5. RESULTS AND DISCUSSION

This section presents the results gathered from the analysis and our observations w.r.t. each research question addressed in this paper.

We use the term "total detected smells (by volume)" to refer to all the smell instances detected in a project. We use the term "total detected smells (by existence)" to refer to the number of different types of smell instances detected in a project. Additionally, we refer to each cataloged configuration smell as a three letter acronym as defined in the Section 3.

Table 3: Distribution of Detected Implementation (ICS) and Design (DCS) Configuration Smells

ICS	#I(V)	#I(E)	DCS	#I(V)	#I(E)
IMD	4,604	706	DMF	64,266	4,339
INC	4,804	440	DUA	4,319	1,427
ICE	3,994	963	DIA	4,354	1,575
IDE	65	29	DMA	1,913	813
IMA	22,976	1,383	DIM	96,033	4,422
IIA	780,265	3,064	DUM	4,653	3,337
IPV	14,360	729	DDB	17,601	1,016
IIT	11,071	1,467	DBH	83	37
IDS	6,466	674	DDS	1760	1760
IQU	428,951	2,463	DDE	1,075	424
ILS	527,637	4,115	DWM	13,944	2,890
IIC	4,797	1,217			
IUV	71339	1,405			

RQ1. *What is the distribution of maintainability smells in configuration code?*

Approach: We compute the total number of detected smell instances (by volume and by existence) for all the smells belonging to both implementation and design configuration smells categories.

Results: The left pan of Table 3 shows the number of smell instances detected for implementation configuration smells (ICS) both by volume (I(V)) and by existence (I(E)). The three most frequently occurring smells by volume and by existence are IIA (*improper alignment*), IQU (*improper quote usage*), and ILS (*long statement*). Similarly, IDE (*duplicate entry*), IMD (*missing default case*), INC (*inconsistent naming convention*) are some of the least frequently occurring smells.

The right pan of Table 3 shows the similar distribution for detected design configuration smells (DCS). The most frequently occurring design configuration smells are DIM (*insufficient modularization*) and DMF (*multifaceted abstraction*). Similarly, the least occurring smells are DBH (*broken hierarchy*) and DDE (*deficient encapsulation*).

A few observations from the above table are the following.

- There is a relatively large number of smell instances reported for DDB (*duplicate block*) by volume; however, the smell only occurs in less than one forth of the analyzed repositories. **This indicates that either the developers of Puppet repositories do not duplicate the code at all or they do it massively.**

- Although, investigating and establishing the potential reasons of identified smell instances is not in the scope of this study, the nascent maturity phase of current configuration systems could be a cause for a few smells. Specifically, the support for system configuration code in terms of better tools and IDEs is still maturing which could potentially avoid smells such as IIA (*improper alignment*).

  The reported frequently occurring smells may also motivate efforts in the future to identify their causes and steps to avoid them. Such efforts may focus on improving existing documentation, enhancing language support, and developing new tools.

- It is interesting to note that DDS (*dense structure*) falls in the least occurring smell category by volume but most frequently occurring smell category by existence. It is due to the fact that there could be only one instance at the most for this smell in a project.

  DUM (*unstructured module*) also exhibits similar characteristics. Since the tool analyzes the structure of a module as a whole, DUM gets identified at the most once for a module. Since each project deals with only a limited number of modules, the detected smell instances are relatively low whereas the smell occurred in relatively large number of analyzed projects.

RQ2. *What is the relationship between the occurrence of design configuration smells and implementation configuration smells?*

Approach: We count the total number of implementation and design configuration smells for each Puppet repository both by volume and by existence. Next, we compute Spearman's correlation coefficient between the counted implementation and design configuration smells for each repository by volume and by existence.

Results: Figure 2 presents a scatter graph (with alpha set to 0.3) showing the co-occurrence between implementation and design configuration smells by volume. The figure shows a dense accumulation towards the left-bottom indicating that implementation and design configuration smells co-occur together for relatively small number of identified smell instances.

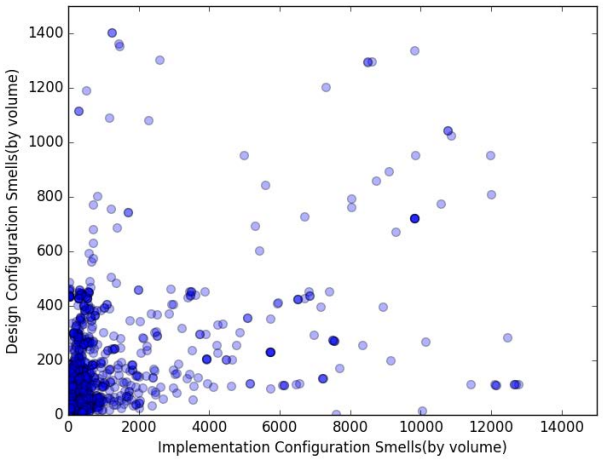

Figure 2: Co-occurrence between implementation and design configuration smells (by volume)

Figure 3 shows a density graph showing the co-occurrence between implementation and design configuration smells by existence. The figure reveals a dense correlation between implementation and design configuration smells (by existence) in the left bottom quadrant of the figure where the number of identified smell types is half or less.

We compute Spearman's correlation coefficient for both the cases. Table 4 shows the correlation coefficients and associated p-value. Both the analyses show positive correlation between implementation and design configuration

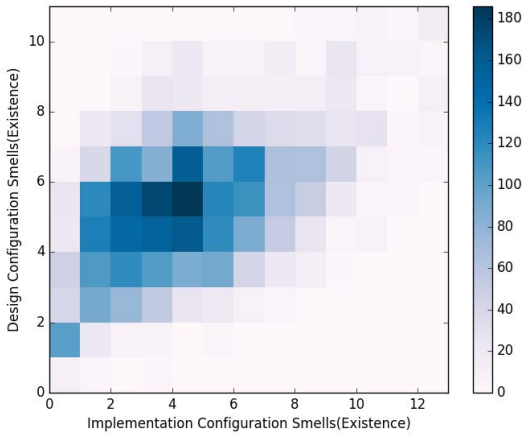

Figure 3: Co-occurrence between implementation and design configuration smells (by existence)

Table 4: Results of Correlation Analysis

	Correlation(ρ)	p-value
Analysis by volume	0.66410	<2.2e−16
Analysis by existence	0.44526	<2.2e−16

smells with high statistical significance; however, correlation analysis by volume exhibits stronger correlation than correlation analysis by existence. **It shows that high volume of design (or implementation) configuration smells is a strong indication of the presence of high volume of implementation (or design) configuration smells in a project.** Whereas, a project that shows presence of large number of design (or implementation) configuration smell types moderately indicates presence of large number of implementation (or design) configuration smell types.

RQ3. *Is the principle of coexistence applicable to smells in configuration projects?*

Approach: To compute intra-category co-occurrence for a smell, we count the number of occurrences of other smells in the same category (by existence), only when the smell occurred. We evaluate the average co-occurrence for each smell across all the repositories considering only those values where the smell has occurred. We compute the average co-occurrence for all the implementation and design configuration smells and compared their normalized values.

Results: Figure 4 presents the average co-occurrence computed for each smell for both the implementation and design configuration smells. IDE (*duplicate entity*) with average 0.75 and IQU (*improper quote usage*) with average 0.29 are the implementation configuration smells that show the highest and lowest co-occurrences in the category. In the design configuration smells category, DBH (*broken hierarchy*) with average 0.73 and DIM (*insufficient modularization*) as well as DMF (*multifaceted abstraction*) with average 0.36 show the highest and lowest co-occurrences respectively.

It means that whenever *duplicate entity* or *broken hierarchy* smells are found, it is very likely to find other smells from the same category in the project.

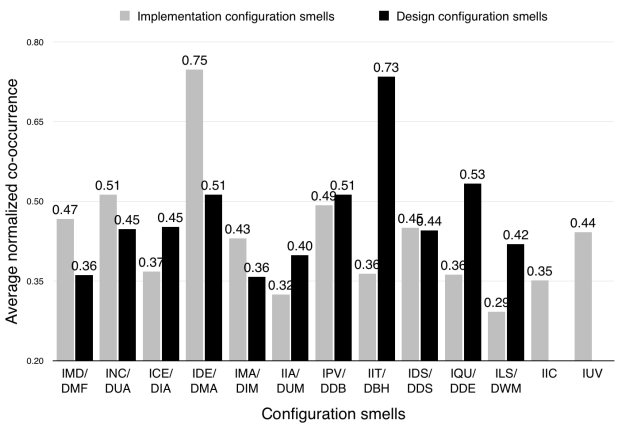

Figure 4: Average co-occurrence (intra-category) for implementation and design configuration smells

Whereas, the smells *improper quote usage* and *insufficient modularization* occur more independently.

Average normalized correlation for implementation and design configuration smells is 0.43 and 0.47 respectively. This leads to another interesting observation: **design configuration smells show 9.3% higher co-occurrence among themselves than the implementation configuration smells.** Since a design decision impacts the software in many ways, it is believed that one wrong or non-optimal design decision introduces many quality issues. The statistic reported above affirms the belief.

RQ4. *Does smell density depend on the size of the configuration project?*

Approach: We compute normalized smell density for both the smell categories for all the repositories and plot scatter graphs between lines of code in the repository and the smell density. We then perform correlation analysis on both the sets and document our observations based on the received results.

Results: Figure 5 and Figure 6 present the distribution of normalized smell density for implementation and design configuration smells against lines of code (with alpha = 0.3).

The visual inspection of the above graphs reveal the following observations.

- The distribution shown in Figure 5 is very scattered and random in comparison with the distribution in Figure 6. Although, both the figures show a weak linear relationship between smell density and lines of code, Figure 6 shows a relatively stronger linear relationship than Figure 5.

- Large projects show maturity and tend to demonstrate lower smell density compared to smaller projects in both the cases.

We, then, computed Spearman's correlation coefficient for both the data sets. The results of the correlation analysis are summarized in Table 5.

The results show no correlation between implementation smell density and size of the project. However, a weak negative correlation is perceived with high statistical significance

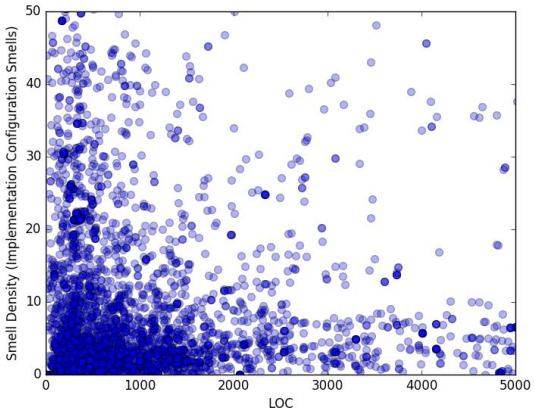

Figure 5: Smell density for implementation configuration smells against lines of code

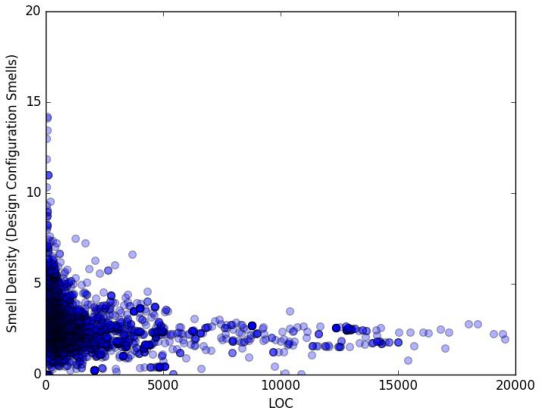

Figure 6: Smell density for design configuration smells against lines of code

Table 5: Results of Correlation Analysis

	Correlation(ρ)	p-value
Implementation smells	0.03833	0.00980
Design smells	-0.32851	<2.2e−16

between design smell density and size of the project. **It shows that as the project size increases, design configuration smell density tends to decrease.** This result is interesting since it is believed traditionally that the complexity (and therefore, smell density) of a piece of software increases as size of the software grows.

We observe another interesting aspect related to configuration smells. Our empirical study reveal a large number of class declarations where the corresponding definitions are not found in the same repository. We find 59% of the repositories that we analyze have at least one such instance. Majority of such missing definitions relate to third-party modules. A possible explanation is that software development teams exclude third-party modules from their Puppet code under version control. This practice provides the advantage of not having to maintain the used third-party modules as they change. However, it breaks the fundamental principle of IaC, i.e. production and configuration code should coevolve. Such instances hurt the configuration process's automation and are bound to lead to trouble in the form of missing dependencies. More interestingly, the Puppet language does not offer any solution to this problem since module installation, as opposed to package installation, cannot be part of a configuration code specification.

Yet another observation concerns language design (in this context Puppet). Diligent use of language features and adherence to best practices can drastically reduce smells discussed in this paper. However, careful language design can also significantly avoid many configuration smells. For example, DUA (*unnecessary abstraction*), DBH (*broken hierarchy*), and DDE (*deficient encapsulation*) can be controlled and avoided by suitable changes in the Puppet language. Similarly, many implementation configuration smells such as IUV (*unguarded variable*), IMA (*misplaced attribute*), and IDE (*duplicate entity*) can also be checked at Puppet language level itself and can be avoided without any compromise in functionality and convenience.

6. RELATED WORK

Our work is related to studies of code quality practices in traditional software engineering and system configuration management.

6.1 Code Quality Practices in Traditional Software Engineering

Fowler [8] characterized code smells as poor design and implementation decisions and identified numerous code and design smells. Girish et al. [36] provided a comprehensive catalog of structural design smells classified based on the principle that they violate. Similarly, Garcia et al. [9] provided a catalog of smells that may arise at architectural level. Many of our cataloged smells have names similar to ones in the literature [8, 36]; however, their meaning and context is aligned to the configuration domain. Additionally, we have added a few new ones; for example *unstructured module* and *dense structure*.

Smells lead to technical debt and affect maintainability negatively [15]. A popular approach for detecting them is static code analysis, which we also employ in this work. Metrics-based rules [19] identify code smells, such as God class or blob, by comparing computed metrics to specified thresholds. Decor [22] provides a domain specific language for formulating rules that detect smells such as blob, functional decomposition, and swiss army knife. Designite [30] detects numerous implementation and design smells. Along the similar lines, various tools to identify refactoring candidates have been proposed; for example, extract method refactoring [38, 29] and extract class refactoring [31, 7].

In addition to the above approaches that base their analysis on a given code snapshot, HIST [23] and Clio [40] consider change history information. HIST [23] analyses changes between software components to detect blob, divergent change, shotgun surgery, parallel inheritance, and feature envy smells. Clio [40] detects software modularity violations with the study of co-evolving software components throughout project history. Our tool *Puppeteer* also relies on static code analysis and reveals design configuration smells.

6.2 Code Quality Practices in System Configuration Management

In the landscape of system configuration management, empirical studies on configuration code written in languages such as Puppet [39] and Chef [37] are scarce. Jiang *et al.* [14] study the co-evolution of Puppet and Chef configuration files with source, test, and build code. They analyze the software repositories of 256 OpenStack projects and distinguish files as infrastructure, which contain configuration code in Puppet or Chef language, production, build, and test. They find that configuration code comes in large files, changes more frequently, and presents tight coupling with test files. In our study, we perform static analysis on Puppet configuration code using three different tools (Puppet-Lint, PMD-CPD, and Puppeteer) to detect configuration smells in a large number of repositories.

Puppet Forge [25] — the repository of Puppet modules, provides an evaluation of configuration code quality through a quality score based on three aspects: code quality score provided by Puppet-Lint [27], compatibility with Puppet, and metadata quality. Metadata quality is subject to a set of guidelines that metadata files should adhere to.

Sonar-Puppet [24] is a SonarQube [32] plug-in that has numerous rules to detect quality violations in Puppet code; most of the rules applied by Sonar-Puppet are common with Puppet-Lint. We have included Puppet-Lint and the additional rules checked by Sonar-Puppet in our analysis and mapped the rules to implementation configuration smells. We have also implemented Puppeteer that identifies all the cataloged design configuration smells.

Various authors have published their ideas describing best practices for configuration code in the form of blog-posts, articles, and technical talks [16, 17, 35, 18]. However, to the best of our knowledge, our study is the first attempt towards formalizing and detecting smells in configuration systems.

7. THREATS TO VALIDITY

Construct validity concerns the appropriateness of observations made on the basis of measurements. False positives and false negatives are always associated with static code analysis and so are applicable to the tools that we used or developed for this experiment. However, the effect of false positives and negatives is reduced when two or more streams of results are compared as in this experiment. A comprehensive set of test cases can rule out obvious deficiencies. Therefore, we employed a comprehensive set of test-cases for Puppeteer.

Similarly, chosen threshold values in any smell detection tool play an important role since their values decide the volume of detected smell instances. Many authors such as Linda et al. [28] have suggested the chosen threshold values after careful analysis. We carefully chose threshold values by adapting commonly used thresholds in traditional software engineering within IaC paradigm in such a way that they are neither too lenient nor very stringent.

Further, a source-code analysis may adopt one of the numerous techniques to collect source-code information such as AST parsing, reflection, and string matching [34]. Due to the lack of available parser library for Puppet, our tool uses regular expression based string matching extensively which is not as efficient as AST parsing. In addition to test the tool with unit-tests that check the correctness of the used regular expressions, we carried out manual testing to ensure the behaviour of the used expressions.

External validity concerns generalizability and repeatability of the produced results. Our experiment analyzes only Puppet repositories whereas there are many other configuration management systems. Although, the employed tools are specific to one configuration language, the proposed theoretical model is general and language agnostic. We believe that it will open doors for similar studies for other configuration management systems.

8. CONCLUSIONS AND FUTURE WORK

The paper presents a preliminary quality analysis for configuration code. We propose a catalog of 13 implementation and 11 design configuration smells based on commonly known best practices. We analyzed 4,621 Puppet repositories containing 142,662 Puppet files and more than 8.9 million lines of code. We investigated four research questions using smell instances detected by our analysis. The key findings from our analysis are the following.

- The developers of Puppet repositories either do not introduce code-clones at all or they do it in a massive scale. Code-clones and other frequently occurring smells may motivate studies in the future to improve support for avoiding and refactoring such smells.
- Configuration smells belonging to a smell category tend to co-occur with configuration smells belonging to another smell category when correlation is computed by volume of identified smells.
- Design configuration smells show 9% higher average co-occurrence among themselves than the implementation configuration smells. This observation affirms the belief that one wrong or non-optimal design decision introduces many quality issues and therefore suggests the developers to take design decisions critically and diligently.
- Design configuration smell density shows negative correlation whereas implementation configuration smell density exhibits no correlation with size of a project.

The study brings benefits for both the researcher and practitioner community. It offers a method for investigating system configuration smells by mining repositories which aims to benefit the research community. At the same time, practitioners can identify configuration smells using the tools employed in this study and adopt best practices to write maintainable configuration code.

Our future research may take many possible directions including investigating various aspects relevant to quality when configuration repositories evolve, applying automated refactoring to configuration code, and measuring the impact of smells on various system parameters.

Acknowledgements

This work is partially funded by the SENECA project, which is part of the Marie Skłodowska-Curie Innovative Training Networks (ITN-EID). Grant agreement number 642954.

We would like to thank Faidon Liambotis (Principal Operations Engineer at Wikimedia Foundation) for providing helpful inputs on our early configuration smells catalog.

9. REFERENCES

[1] Ansible. http://www.ansible.com/, 2016. [Online; accessed 22-Jan-2016].

[2] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *34th International Conference on Software Engineering*, pages 419–429, June 2012.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition, 1996.

[4] CFEngine. https://cfengine.com/, 2016. [Online; accessed 22-Jan-2016].

[5] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, Jun 1994.

[6] PMD—CPD: Copy Paste Detector. https://pmd.github.io/, 2016. [Online; accessed 22-Jan-2016].

[7] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. JDeodorant: Identification and Application of Extract Class Refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1037–1039, New York, NY, USA, 2011. ACM.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 edition, 1999.

[9] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a Catalogue of Architectural Bad Smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, QoSA '09, pages 146–162, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] GitHub. https://github.com/, 2016. [Online; accessed 22-Jan-2016].

[11] G. Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[12] G. Gousios and D. Spinellis. GHTorrent: Github's data from a firehose. In *9th IEEE Working Conference on Mining Software Repositories*, pages 12–21, June 2012.

[13] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1 edition, 2010.

[14] Y. Jiang and B. Adams. Co-evolution of Infrastructure and Source Code: An Empirical Study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 45–55, Piscataway, NJ, USA, 2015. IEEE Press.

[15] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, 2012.

[16] G. Larizza. Building a Functional Puppet Workflow Part 1: Module Structure. http://www.webcitation.org/6g23RY7yS, 2016. [Online; accessed 15-Mar-2016].

[17] G. Larizza. Building a Functional Puppet Workflow Part 2: Module Structure. http://www.webcitation.org/6g23YeuFl, 2016. [Online; accessed 15-Mar-2016].

[18] G. Larizza. Doing the Refactor Dance — Making Your Puppet Modules More Modular. http://www.webcitation.org/6g23dnNKo, 2016. [Online; accessed 15-Mar-2016].

[19] R. Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[20] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 1 edition, 2002.

[21] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 1 edition, 2008.

[22] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions of Software Eng.*, 36(1):20–36, 2010.

[23] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *IEEE/ACM 28th International Conference on Automated Software Engineering*, pages 268–278, Nov 2013.

[24] S. Puppet. SonarQube Puppet Plugin, Last accessed on: 22nd Jan 2016. Available at: https://github.com/iwarapter/sonar-puppet.

[25] Puppet Forge: a repository of Puppet modules, Last accessed on: 22nd Jan 2016. Available at: https://forge.puppetlabs.com.

[26] Puppet Labs. https://puppetlabs.com/, 2016. [Online; accessed 22-Jan-2016].

[27] Puppet-lint: Puppet code style checker, Last accessed on: 22nd Jan 2016. Available at: http://puppet-lint.com.

[28] L. H. Rosenberg, R. Stapko, and A. Gallo. Risk-based object oriented testing. In *Twenty-Fourth Annual Software Engineering Workshop*, Greenbelt, MD, Dec. 1999. NASA, Software Engineering Laboratory.

[29] T. Sharma. Identifying Extract-method Refactoring Candidates Automatically. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 50–53, New York, NY, USA, 2012. ACM.

[30] T. Sharma, P. Mishra, and R. Tiwari. Designite — A Software Design Quality Assessment Tool. In *Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities*, BRIDGE '16, New York, NY, USA, To appear. ACM.

[31] T. Sharma and P. Murthy. ESA: The Exclusive-similarity Algorithm for Identifying Extract-class Refactoring Candidates Automatically. In *Proceedings of the 7th India Software Engineering Conference*, ISEC '14, pages 15:1–15:6, New York, NY, USA, 2014. ACM.

[32] SonarQube. http://www.sonarqube.org/, 2016. [Online; accessed 22-Jan-2016].

[33] D. Spinellis. Version control systems. *IEEE Software*, 22(5):108–109, September/October 2005.

[34] D. Spinellis. Tools and Techniques for Analyzing Product and Process Data. In T. Menzies, C. Bird, and T. Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 161–212. Morgan-Kaufmann, 2015.

[35] Puppet language style guide. https://docs.puppetlabs.com/guides/style_guide.html, 2016. [Online; accessed 22-Jan-2016].

[36] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt* . Morgan Kaufmann, 1 edition, 2014.

[37] M. Taylor and S. Vargo. *Learning Chef—A Guide To Configuration Management And Automation*. O'Reilly Media, 1 edition, 2013.

[38] N. Tsantalis and A. Chatzigeorgiou. Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods. *Journal of Systems and Software*, 84(10):1757–1782, Oct. 2011.

[39] J. Turnbull and J. McCune. *Pro Puppet*. Apress, 1 edition, 2011.

[40] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting Software Modularity Violations. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 411–420, New York, NY, USA, 2011. ACM.