Does the Choice of Configuration Framework Matter for Developers? Empirical Study on 11 Java Configuration Frameworks

Mohammed Sayagh[1], Zhen Dong[2], Artur Andrzejak[2] and Bram Adams[1]
[1]MCIS, Polytechnique Montreal, {mohammed.sayagh,bram.adams}@polymtl.ca
[2]Institute of Computer Science, Heidelberg University, {artur.andrzejak,zhen.dong}@informatik.uni-heidelberg.de

***Abstract*— Configuration frameworks are routinely used in software systems to change application behavior without recompilation. Selecting a suitable configuration framework among the vast variety of existing choices is a crucial decision for developers, as it can impact project reliability and its maintenance profile. In this paper, we analyze almost 2,000 Java projects on GitHub to investigate the features and properties of 11 major Java configuration frameworks. We analyze the popularity of the frameworks and try to identify links between the maintenance effort involved with the usage of these frameworks and the frameworks' properties. More basic frameworks turn out to be the most popular, but in half of the cases are complemented by more complex frameworks. Furthermore, younger, more active frameworks with more detailed documentation, support for hierarchical configuration models and/or more data formats seem to require more maintenance by client developers.**

I. Introduction

Modern software systems are expected to be highly configurable to satisfy the personal requirements and preferences of their users. For instance, the distributed storage and computing framework Apache Hadoop 2.7.1. has more than 800 configuration options, where the web browser Mozilla Firefox 43.0 has over 2,000 configuration options available to users! The impact of these options ranges from controlling active features, to adjusting the performance, storing GUI preferences or integrating a software product with its runtime environment (e.g., URL of database or web server). Users typically are able to change the value of configuration options via dedicated configuration files, command line parameters or even at run-time. For example, Firefox has a dedicated "about:config" configuration page to change its configuration on-the-fly.

Since the configuration of a software system is not trivial, but has to support different storage formats, offline/online manipulation, etc., developers are not required to develop their own framework, but have access to a wide range of general-purpose, open source configuration frameworks for their programming language of choice. Quick searches for "configuration" in Google or StackOverflow yield dozens of hits for each programming language, varying from tiny one-man configuration frameworks (e.g., Raihan's jconfig [25][1]), frameworks included in the standard library of a language (e.g., Java property files) to large, established configuration frameworks (e.g., Typesafe configuration framework [33]). In addition to this, large communities such as Mozilla Foundation or Apache Foundation developed their own configuration frameworks that have become popular outside as well [1].

This paper analyzes whether the choice of configuration framework actually matters from the point of view of developers. Indeed, various studies [12], [19], [20], [36], [23] have shown how software configuration errors are one of the major causes of today's system failures, with sometimes up to 27% of reported issues labeled as configuration-related [37]. A significant part of such configuration errors are caused by improper design and implementation of configuration-related code and configuration options, such as lack of explicit log messages for configuration errors or lack of type checking of configuration values [36], or the constant need for maintaining too strongly coupled configuration-related source code. Such errors could be avoided by choosing the right configuration framework for a project. Furthermore, depending on how a configuration framework is integrated into a project (modular access vs. high coupling), and the rate of new releases of the framework, developers might need to spend substantial effort maintaining their use of the configuration framework.

Hence, this paper performs an empirical study of the characteristics, popularity and maintenance overhead related to the use of 11 general-purpose Java configuration frameworks by analyzing 1,938 GitHub-based Java projects. The contributions of our work are the following:

- Using manual analysis, we build a taxonomy of the major features of configuration frameworks. This taxonomy helps understand the wide range of features and differences amongst the frameworks.
- We study the popularity of the 11 frameworks by analyzing 1,938 projects sampled from GitHub, and which frameworks are typically used together in a project.
- We build classification models to understand the project and framework characteristics impacting the maintenance effort for using a configuration framework.

The results of this paper will help to understand the amount of attention necessary to select a configuration framework for a client project.

II. Background and Related Work

This section provides background and related work on software configuration and configuration frameworks.

[1]Not to be confused with the jConfig framework studied in this paper [13].

A. Software Configuration Frameworks

Software configuration is the mechanism used to adapt a software system to different contexts, simply by changing the value of certain configuration options. As such changes can be performed by end users, no recompilation is required. For example, a user can change the database used by a software system only by changing the URL, username and password in the associated configuration option.

Such options can be stored in different storage formats and can use different configuration models. Common storage formats are JSON or XML files, SQL databases or (more advanced) distributed configuration databases such as ZooKeeper. Configuration models can range from basic key-value pairs to more elaborate hierarchical structures used for example by the Linux kernel or the Windows Registry, where related low-level options are combined into groups that can be combined recursively with other option groups.

Many configuration frameworks have been proposed to manage configuration options, storage formats, models, and other configuration-related functionality. Such frameworks provide an API for developers to read an option's value within a project's source code, such that software developers do not need to implement their own framework. For each programming language, dozens of open-source configuration frameworks are available, each with its own focus and feature set. For example, one of the simplest Java configuration frameworks is *Java Properties*, which allow to read key-value configuration pairs from textual files. However, many other, 3rd party frameworks exist. In this paper we study their differences and impact on software maintenance.

B. Related Work

As configuration issues contribute 17% of the total cost of ownership of todays software, troubleshooting misconfigurations is a large part of technical support [17]. Given the role of configuration frameworks in avoiding configuration errors, it is important to understand the impact of configuration framework choice for a software project. Several researchers have focused on analyzing and comparing configuration frameworks, although without considering the impact on developers. Moreover, a large body of research exists on configuration errors and debugging.

Configuration Frameworks. Rabkin *et al.* analyze configuration frameworks in 7 large-scale Java software projects and find that all 7 projects used a standard key-value configuration model to organize configuration data. They also made a taxonomy of configuration options, concluding that most configuration options fall into a small number of data types. Differently, our work focuses on the taxonomy of existing configuration frameworks and their properties when used in the application development.

Other research groups studied projects that use a hierarchical configuration model, in which options are organized into a tree instead of a flat key-value model [3], [14]. Such hierarchical configuration data is widely used [14]. Tresch investigated common configuration frameworks for Java, briefly summarizing the application scenarios and configuration data formats for each framework [32], while Denisov summarized the features of three major configuration frameworks (java.util.prefs.Preferences, java.util.Properties and Apache Common Configuration) [9]. Instead of focusing on a handful of configuration frameworks and some of their features, we performed a detailed analysis on 11 configuration frameworks for Java.

TABLE I
THE STUDIED CONFIGURATION FRAMEWORKS AND THEIR SIGNATURES.

#	Framework	Released	Signature
1	Properties	01.1996	import java.util.Properties
2	System	01.1996	System.getProperty
3	jConfig [13]	10.2001	import org.jconfig.*
4	Preferences	07.2003	import java.util.prefs.*
5	Spring [30]	06.2003	import org.springframework.*
6	Commons [1]	11.2005	import org.apache.commons.configuration.*
7	Constretto [7]	05.2010	import org.constretto.*
8	Typesafe [33]	12.2011	import com.typesafe.config.*
9	Deltaspike [8]	10.2012	import org.apache.deltaspike.*
10	Owner [21]	12.2012	import org.aeonbits.owner.*
11	CFG4J [5]	07.2015	import org.cfg4j.*

Configuration Errors. Xu *et el.* [35] study 620 user-reported configuration errors from 4 software projects, i.e., Apache HTTP server, MySQL database, Apache Hadoop and a commercial storage system. All these projects have the over-designed configuration problem, i.e., they offer hundreds of configurations that are unused, but may impact behavior of the system in unforeseen ways if touched incorrectly. For instance, 51.9% options can be removed without impacting the usage of the system. Yin *et al.* [37] also study 546 misconfigurations from four widely used open source systems (CentOS, MySQL, Apache Http Server, and OpenLDAP). Of these, 70% to 85.5% are due to mistakes in choosing the value of configuration options, while another significant number of misconfigurations are due to compatibility issues between different components or modules. Those misconfigurations can be reduced by adopting a well-designed configuration mechanism in software development.

Arshad *et al.* [2] study 281 bug reports related to configuration for the GlassFish and JBoss Java EE application servers. They find that a significant part of configuration errors are due to mistakes by the developers and require code modifications to fix the problem. Sayagh *et al.* [27], [26] studied configuration errors that span across different layers of a software stack such as LAMP. One of the causes of such errors is the diversity of configuration frameworks and models used across layers. Other research [11], [10], [24], [38] has studied the impact of configuration models such as key-value pairs on software misconfiguration. All this work indicates that configuration frameworks play an important role in configuration errors and maintenance, and hence requires careful selection. This paper analyzes this role.

III. TAXONOMY OF CONFIGURATION FRAMEWORKS

In this section, we introduce the 11 Java configuration frameworks that we are studying, as well as the taxonomy of their features and properties that we derived.

TABLE II

TAXONOMY OF THE 11 STUDIED CONFIGURATION FRAMEWORKS (NUMBERED ACCORDING TO TABLE I).

Dimensions	Properties	1	2	3	4	5	6	7	8	9	10	11
General Properties	Universal	✓	✓	✓	✓		✓	✓	✓		✓	✓
	Part of SDK	✓	✓		✓							
	Age	high	high	med	med	med	med	low	low	low	low	low
	Quality of Documentation	high	high	low	high	high	high	med	high	med	high	med
	Actively Maintained	✓	✓		✓	✓	✓		✓	✓	✓	✓
Feature Richness	Multiple Storage Formats	low	low	med	med	med	high	high	med	med	low	med
	Hierarchical Configuration Struct.			✓		✓	✓	✓	✓	✓	✓	
	Hierarchical Overriding			✓		✓	✓	✓	✓	✓	✓	
	Multiple Data Sources			✓	✓	✓	✓		✓			✓
	Variable Substitution					✓	✓		✓	✓	✓	
	#API methods	15	5	273	39	1,681	1,022	191	109	44	37	57
	#Annotations	0	0	0	0	242	0	6	1	16	12	0
Programming Support	Dependencies	none	none	med	none	high	high	med	none	none	none	med
	Distributed Environment Support			✓		✓	✓					✓
	Type-safety			✓	✓	✓	✓	✓	✓	✓	✓	✓
	Notification Mechanisms			✓		✓	✓				✓	
	Configuration Injection					✓	✓	✓		✓	✓	

A. Configuration Frameworks

This paper focuses on general-purpose configuration frameworks, i.e., configuration frameworks that can be used in a variety of software systems, from desktop applications to mobile apps or enterprise software. As such, our analysis is relevant to a wide range of systems. Furthermore, given Java's 20+ years of history and the many (open source) configuration frameworks available for it, we focused exclusively on general-purpose Java configuration frameworks. Other programming languages left for future work.

To obtain the catalog of Java configuration frameworks used in this paper, the first two authors performed search queries using different keywords and phrases such as "Java configuration frameworks" and "Java configuration tools", then read a large amount of technical fora and blogs. They quickly converged on a set of 14 frameworks covering a wide range of mature and young frameworks, which are shown in Table I, ordered based on the date of their first release.

Before considering the properties of these frameworks in more detail, it is important to note that some configuration frameworks were not included in the paper. Play [22] is a web application framework whose configuration framework basically is a modified version of the Typesafe framework. Furthermore, neither Carbon [4], nor Raihan's jconfig [25] had any GitHub project as user, hence we excluded these frameworks as well. Android's SharedPreferences framework [28] was excluded, since it only applies to Android apps, and not to desktop or enterprise applications. Finally, we did include Spring and Deltaspike, since they support desktop and enterprise applications, although they do not support mobile apps. The resulting set of 11 frameworks is used in the remainder of this paper.

B. Taxonomy

In order to understand the differences between configuration frameworks in terms of features and properties, and later relate those to the popularity and maintenance effort involved with the usage of these frameworks, we built a taxonomy of configuration frameworks, then classified each framework according to the taxonomy.

To determine the taxonomy's properties, for each framework at least two of the authors manually studied the public documentation and browsed forums and blog posts associated with each framework. Any relevant property recurring within the analyzed framework was tagged. Then, once all frameworks were analyzed, we compared the tagged properties across all frameworks to arrive at a final list of 17 framework properties, grouped into 3 dimensions. The resulting taxonomy, as well as each framework's classification, is shown in Table II. Note that each framework's tagged properties were checked by two authors, while the full set of properties was obtained by all authors together.

The first taxonomy dimension, i.e., *General Properties*, contains basic information about the configuration frameworks. *Universal* indicates whether a configuration framework is fully general-purpose, or does not support mobile apps. *JDK-Standard* indicates if a framework is integrated into the Java SDK libraries. *Age* is *low* if a framework was created after 2010, *high* if before 2000, *med* if in between 2010 and 2000. If there is no documentation for a framework, we consider its *Quality of Documentation* as *low*. If the documentation is not very comprehensive and/or written in a non-rigorous manner, the quality is considered as *med*. Finally, for comprehensive and concrete documentation, we consider the quality as *high*. To mitigate subjectivity of assessing quality of documentation, for each framework, two persons separately looked for related documents such as JavaDoc and tutorial documents, and had a discussion to decide its value. *Actively Maintained* indicates whether there has been at least one commit to the configuration framework's code repository in 2016.

The second dimension, i.e., *Feature Richness*, measures how powerful the framework is. *Multiple Storage Formats* measures the number of data formats (such as XML, properties files or JSON) a framework supports: *low* (1∼2 formats), *med* (3∼4 formats) and *high* (≥5 formats). *Hierarchical Configuration Structure* indicates if a framework supports hierarchically organized configuration data (e.g., tree structured) or just flat key-value pairs. *Hierarchical Overriding* means that the value of an option configured in a lower

priority layer can be overridden by a higher priority layer. *Multiple Data Source* indicates that a framework is able to load configuration data from multiple sources instead of just from one file. *Variable Substitution* specifies whether the user can define and use variables in the configuration file instead of having to copy repetitive configuration values throughout. *#API methods* is the number of public methods within public classes of each configuration framework. Similarly, *#Annotations* is the number of public Java annotations proposed by each of the studied configuration frameworks. The last two metrics are basically obtained by using the JavaParser tool. Note that for all these frameworks, methods that are within test classes are ignored. In addition, since Deltaspike and Spring contain more than just configuration functionality, we consider for these two frameworks only the classes whose own name or package name contains the keyword *config*.

The final dimension, *Programming Support*, contains properties supporting programmers when integrating the configuration framework in their source code. *Dependencies* measures how many dependent libraries need to be imported before using a configuration framework: *none* (0), *med* (0$\sim$10), and *high* ($\geq$10). *Distributed Environment Support* indicates if a framework can be used in a distributed setting, for example through the use of a configuration database. *Type-safety* indicates that a framework checks whether the value of a configuration option has the right type (e.g., double vs. integer) when read. *Notification Mechanisms* specifies that a system will get a notification from the configuration framework when configuration data has been changed by the user. Finally, *Configuration Injection* allows configuration to be fully defined in external files, with the corresponding configuration values automatically injected in the source code.

Apart from the three frameworks included in the SDK (Properties, System and Preferences), all frameworks cover a wide range of features, which confirms the need for a study like this paper! Only Type-safety is shared amongst all non-SDK frameworks, but no other clear pattern of feature usage can be found. The most rare features are Distributed Environment Support and Notification Mechanisms, with jConfig, Spring and Commons supporting both of these. The latter two frameworks are the most fully featured, followed by Owner and jConfig.

IV. Collected Data

Now that we understand the different features of the configuration frameworks, we aim to study the popularity of each framework (Section V) as well as any relation between framework or project features and the amount of maintenance effort required in projects using those frameworks (Section VI). Each of these analyses uses a different data set of GitHub projects, which we will refer to as "Data Set 1" (popularity) and "Data Set 2" (maintenance effort).

A. Data Set 1: Popularity

Sampling. We first used GHTorrent to create a list of all non-fork Java projects on GitHub with at least 50 commits. The latter constraint is important to eliminate as many toy projects as possible, or repositories that are just mirrors (and hence only have few GitHub commits) [16]. Then, we randomly sampled and cloned 10,000 projects from the list. To verify the diversity of this sample, we used the approach of Nagappan et al. [18] to compute a *diversity score*. Our diversity score considered the following metrics related to project maturity: number of commits, number of authors, number of committers, and active age (time span between first and last commit). The overall diversity score of 1.00 indicates that our sample is sufficiently diverse.

Despite the high diversity score, the sample still contained many repositories not used for developing software projects [16], such as experimental repositories and repositories containing example code snippets, demos, personal test code, and so forth. Since those repositories are not used to develop software projects, we tried to exclude them from the sample, by selecting only repositories that have at least 3 stars. This yielded Data Set 1, which contains 1,938 repositories (Table III). Note that the number of stars was not available from the GHTorrent database, and hence could only be obtained by scraping the projects' GitHub repositories (using Christophe et al.'s crawler [6]).

Mapping projects to configuration frameworks. To learn which configuration framework(s) is/are used by a project, we automatically scan the projects' source code for *framework signatures*. These are statements indicating that the particular configuration framework is used in the code, see Table I. For most frameworks, the signature is a set of import statements that we manually extracted from the javadoc documentation of the corresponding framework. Only for System Properties the signature is not an import statement (since no import is necessary to use this framework), but instead we search directly for the *System.getProperty*() calls. After scanning the projects of Data Set 1 for these signatures, for each project we end up with zero or more corresponding frameworks being used in the latest snapshot.

B. Data Set 2: Maintenance Overhead

To study maintenance overhead involved with configuration framework usage, we cannot use Data Set 1, since due to the sampling used we might not have sufficient projects for each configuration framework. Instead of random sampling, for Data Set 2 we used the frameworks' signatures to search for projects using each framework in turn.

Querying. To select Java projects that use at least one of the 11 Java configuration frameworks, we used the GitHub search function with a configuration framework signature as a search keyword, for one framework at a time. Since GitHub's web search is limited to 100 pages of search results, each containing 10 files, for each framework we obtained a maximum of 1,000 pages matching the framework's signature. To increase this number, we performed each search three times, since GitHub offers three different ranking algorithms for its search results (*"Best Match, Recently indexed, and Least recently indexed"*). Only Constretto and

TABLE III

POPULARITY OF CONFIGURATION FRAMEWORKS IN DATASET 1. COLUMNS "#/% PROJECTS" REPORT THE NUMBER/PERCENTAGE OF PROJECTS USING A GIVEN FRAMEWORK (PROJECTS MAY USE MULTIPLE FRAMEWORKS), WHILE COLUMN "# 1 CF" SHOWS THE NUMBER OF PROJECTS USING *only* THE GIVEN FRAMEWORK.

Framework	# Projects	% Projects	# 1 CF
System	821	42.36	347
Properties	600	30.96	142
Spring	91	4.70	38
Preferences	57	2.94	6
Commons	37	1.91	5
Typesafe	14	0.72	4
Deltaspike	3	0.15	0
CFG4J	1	0.05	0
Constretto	1	0.05	0
jConfig	0	0	0
Owner	0	0	0
Any Framework	**1,034**	**-**	**542**

TABLE IV

#PROJECTS IN DATASET 2.

CF	# Repositories	# Non-fork
System	889	338
Commons	836	442
Spring	754	291
Properties	744	399
Preferences	662	408
Typesafe	659	383
Deltaspike	232	157
Owner	144	82
jConfig	53	17
Constretto	37	24
CFG4J	13	7
Total	**5,216**	**2,575**

CFG4J yielded less matches than the maximum number of search results.

Using the GitHub search webscraper of Christophe *et al.* [6], we obtained the GitHub project names mentioned on the 100 pages of search results, and after removing projects that are forks, we obtained the projects summarized in Table IV. We scraped their GitHub pages for repository-related meta-data (like number of releases and stars), and also cloned them to analyze their code change history.

Mapping projects to configuration frameworks. We used the same approach as for Data Set 1 to identify the configuration frameworks used by each project in Data Set 2.

V. POPULARITY OF CONFIGURATION FRAMEWORKS

In this section, we study the popularity of configuration frameworks in Github Java projects by addressing two research questions.

RQ1: How Popular are Individual Configuration Frameworks?

Motivation. Although Table II contains a wide variety of publicly available configuration frameworks, many of them have comparable features, for example support for multiple formats of persistent storage, variable substitution, or type safety. Consequently, selecting a configuration framework suitable for a specific Java project typically requires a time-consuming evaluation of alternatives. By assuming that popularity of a framework might be an indication of its maturity and quality, a study of configuration framework popularity can provide hints for developers about which configuration frameworks (and hence features) to prefer.

Approach. For each project in Dataset 1, we identify which configuration framework (or frameworks) it was using at the time of writing this paper. From this information, we compute the popularity statistics of Table III. In particular, we calculate two metrics to measure the popularity of each framework: the number of projects using this configuration framework, and the number of projects *only* using this configuration framework at the time of writing this paper.

Results. Out of the 1,938 projects of Dataset 1, 1,034 projects use one or more configuration frameworks. The popularity of each framework is shown in Table III.

Finding 1: System Properties and (java.util) Properties are the most widely used frameworks. There are 821 projects in the popularity dataset that use the System Properties framework. Similarly, the Properties framework is widely used as well, namely by 600 projects. This result of course comes as no surprise, since these configuration frameworks are integrated in the Java SDK and hence can be directly used without importing any external libraries. However, as we could see in Table II, these are also the weakest frameworks in terms of features.

Finding 2: Third-party configuration frameworks are not that commonly used. Table III shows that the top three third-party frameworks are Spring, Commons, and Typesafe with 91, 37 and 14 projects using them, respectively (Preferences is also included in the Java SDK). Given that Dataset 1 comprises 1,938 projects, the proportion of projects using third-party configuration frameworks is surprisingly low, below 5%. In fact, for some of the third-party frameworks we did not find any project using it in Dataset 1 (which is why we are using the framework-specific Dataset 2 later in this paper). This is partly due to the younger age of those frameworks.

Finding 3: Among the 904 projects without a configuration framework, 362 projects are Android projects. We found that within the considered 904 projects without a framework there are 362 (40.1%) Android projects. Since the Android platform provides a SharedPreferences framework for storing key-value pairs as well as other mechanisms such as support for SQLite databases and XML files, 209 out of 362 Android projects use the SharedPreferences framework, refraining from using a general-purpose configuration framework. The latter statement is confirmed by the fact that among all 470 Android projects in the whole sample (of 1,938 projects), only 108 or 23% use one of the studied configuration frameworks

Of the remaining 542, i.e., non-Android, projects without a configuration framework, we randomly sampled 53 projects (10%) for manual analysis. We found that 26 projects in this sample do not use any configuration mechanism. Among these, there were 7 projects containing code examples (e.g., for interviews or exercises), 6 libraries, 4 plugins, 4 small games, and 4 simple applications. Another 27 projects in the sample use either an ad hoc configuration mechanism (mostly XML files), or are plugins for other applications having their

own configuration mechanism. Only 17% of these 27 projects are stand-alone applications.

Finding 4: Developers prefer easy-to-use and simple configuration frameworks with good documentation, especially Spring, Typesafe, Commons and Properties. We contacted Java developers via 2 Reddit and 4 Facebook groups with a small survey [31] to better understand the criteria considered by developers to choose a suitable configuration framework (RQ1).

From the 10 replies that we received, we learnt that ease-of-use is the developers' primary criterion for framework selection (4 votes), followed by the simplicity (2 votes), quality of documentation (2 votes) and capability of hierarchical overrides (2 votes). In terms of framework recommendations, Spring was rated highest (4 votes), followed by Typesafe, Commons, and Properties (3 votes each). This explains why standard frameworks like Properties are popular, a finding that was confirmed even more by the following quote: *"Think of building software as digging a hole, and the JDK is your shovel. If you are just digging a hole to plant a tree, your shovel will do fine. If you are digging a swimming pool, then you will want to bring in some heavy machinery (aka 3rd party libraries)"*.

RQ2: How Often are Configuration Frameworks Used Together?

Motivation. We found that some projects are using multiple configuration frameworks at the same time. Given the different focus of frameworks in terms of features (see Table II), this could suggest that some frameworks are complementary and serve different purposes. This RQ aims to understand how common such co-occurrences are as well as which frameworks co-occur often.

Approach. Using Dataset 1, we count how many projects use k configuration frameworks in their most recent Git snapshot), for $k = 1, \ldots, 5$. We also investigate whether there is a relationship between project maturity and the number of configuration frameworks a project uses, since older projects might be larger and hence have heavier demands for configuration frameworks. We measure maturity in terms of different metrics, such as age of the project, number of authors, number of commits, number of committers and number of source code files.

Finding 5: 47.5% of the projects using a configuration framework (491 out of 1,034) combines multiple frameworks. 52.5% (543 out of 1,034) of the projects use one framework, with 38.5% (389 out of 1,034) using two configuration frameworks. There are substantially less projects with 3 or more configuration frameworks: 8.3% (86 out of 1,034) with 3 frameworks, and less than 0.6% with 4 or 5 frameworks (6 and 1 projects, respectively).

Finding 6: System and Properties co-occur the most with other frameworks. The co-occurrence heatmap in Figure 1 confirms that combinations of System with other frameworks dominate the co-occurrence relations, followed by Properties and (at some distance) Spring, Preferences and Commons. This ranking matches with the popularity

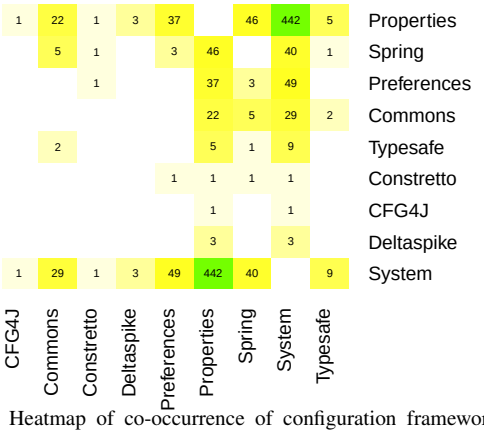

Fig. 1. Heatmap of co-occurrence of configuration frameworks in the projects using a configuration framework.

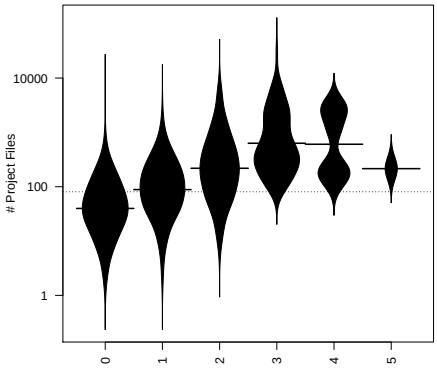

Fig. 2. Bean plots of the number of files (y-axis) within projects, grouped by the number of configuration frameworks used by projects (x-axis).

numbers in Table III, where more popular frameworks co-occur more often with other frameworks.

As mentioned before, System and Properties only provide a limited data set, specializing respectively in access to shell environment variables and to flat files with key-value pairs. For System, no explicit imports are needed and option values can be set via the Java command line, lowering the barrier for using it. This positions both configuration frameworks as an easy-to-use complementary configuration mechanism.

Finding 7: More mature projects tend to use multiple configuration frameworks. The beanplots[2] in Figure 2 show that the number of frameworks used by a project is higher for larger projects (there are only few projects with 4 or more configuration frameworks, so these beans are not significant). Other metrics such as active project age confirmed this finding, which confirms our earlier hypothesis about older projects. We could not find any more complex pattern explaining co-occurrence of configuration frameworks.

Finding 8: The surveyed developers use multiple configuration frameworks in the same project because their library dependencies needed them, or because of complementary features provided by configuration frameworks. Only 4 of the developers in the survey were familiar with projects using multiple configuration frameworks. One of the main explanations for co-occurrence of configuration frame-

[2]A beanplot is a boxplot that also shows the density of the data instead of just a rectangle.

works was dependence on library or component, for example, one developer would add an additional configuration framework*"only if required by a library/framework"*. Another common reason was to exploit the different functionalities of each configuration framework, prompting people to just add a new configuration framework that offers the required features, *"No need to reinvent the wheel for all things. Just add what is missing on top"*.

VI. Maintenance Overhead of Frameworks

In previous sections, we compared different Java configuration frameworks in terms of features and studied their popularity across large and popular Github Java projects. We also qualitatively found that people tend to choose easy-to-use and simple configuration frameworks. In this section, we identify the taxonomy and software project-related properties that define such simplicity, in terms of effort required by client developers to maintain a configuration framework.Those properties are the ones a client developer should consider in order to choose a suitable configuration framework. Our findings could also help configuration framework developers to improve their frameworks by focusing on more critical attributes from a maintenance point of view.

We address the following research questions:

RQ3 *Which factors impact the percentage of configuration-related commits?*

RQ4 *Which factors impact the percentage of configuration-related source files?*

RQ5 *Which factors impact the percentage of developers touching configuration code?*

A. Case Study Setup

The goal of RQ3 to RQ5 is to compare configuration frameworks (in terms of their features) based on the effort required by a developer to maintain them. Maintenance here refers to any changes involving the configuration framework API that a developer needs to do while evolving his or her own software project, for example to update the code to a new version of the framework or to spread configuration data throughout the application.

To perform our analysis, we build classification models explaining one of three effort measures in terms of configuration framework (Table II) and project-related properties. First, we discuss the dependent and independent variables of the models, before discussing how we built and evaluated the models.

Dependent Variables. Table V shows the three dependent variables (CFCommits, CFAuthors and CFFiles), together with three auxiliary variables they are based on. Each dependent variable measures some facet of maintenance effort. As we are building classification models, the percentages of CFCommits, CFAuthors and CFFiles are discretized into a binary value, using the median as threshold. So, all projects whose percentage of configuration framework-related commits is higher than the median of that percentage across all studied projects are considered as having a "high" value, while others have a "low" value.

To determine when a configuration framework is added (*CFAddedIn*) and removed (*CFRemovedIn*), we identified the commits that add or remove signature instances of a configuration framework throughout a project's git repository. By counting the number of "live" signature instances (+1 for each added instance, -1 for each removed one), we can determine the full removal of a framework if the count hits zero.

However, finding all configuration-related commits, i.e., not just those adding or removing the signature (import) of a framework, is less straightforward. One way to find these commits is to analyse each modified line of a commit for calls to one of the methods of a configuration framework API. However, this approach is not accurate, because the number of methods of a configuration framework can be very large, while configuration framework API methods do not always have unique names.

Therefore, we used a technique similar to the approach used by Zhang et al. [39], which consists of selecting as "configuration commits" those that contain in their commit message one of the following keywords related to configuration: *config, property, properti, pref, option,* and *setting*. Hence, for each file containing an import (signature) statement of a configuration framework, we consider its "configuration commits" as the commits related to that configuration framework that we need to analyze. We can then calculate the variable *CFCommits*, while the number of unique configuration framework authors is used for the *CFAuthors* variable.

From the same historical data set, we know all the files that contained code calling the configuration framework, across the whole history of each repository. This enabled us to calculate the total number of these source files across time as the variable *CFFiles*.

Project-related Variables. A first set of independent variables is formed by project-related variables, i.e., variables that control for the activity and state of a source code project as a whole. They are shown at the bottom of Table V. We obtained the set of metrics *Watch, Star, Branch, Releases* by scraping each Github project's web page, then used the git repository's logs to extract the metrics related to the number of files, authors, and commits, which are respectively *NbreFiles, NbreAuthors,* and *NbreCommits*.

Configuration Framework Variables. These variables are the 17 configuration framework properties of Table II. Note that all projects using the same framework share the same value for these properties, only their project-related variables and dependent variables differ. Conversely, projects that at the time of writing this paper use multiple configuration frameworks, were included once for each configuration framework, in which case these data instances had overlapping project-related metrics, but different configuration framework properties.

Building the logistic regression models. To build our logistic regression models [15], our initial training set consists of all projects in Dataset 2 (Table IV). However, since the values of the configuration framework properties are

TABLE V

AUXILIARY, DEPENDENT AND PROJECT-RELATED (CONTROL) VARIABLES CONSIDERED IN THE MAINTAINABILITY MODELS.

Category	Metrics	Description
Auxiliary Variables	CFAddedIn	Date when a configuration framework is added in a project.
	CFRemovedIn	Date when a configuration framework is removed from a project.
	TotalCommitsInCFPeriod	#Commits between the adoption and removal date of a framework in a given project.
Dependent Variables	CFCommits	Percentage of commits touching files that access a given configuration framework.
	CFAuthors	Percentage of authors that change files that access a given configuration framework.
	CFFiles	Percentage of source code files of a project that access a configuration framework.
Project-related (control)	Watch	Number of watches of a Github project.
	Star	Number of stars of a Github project.
	Branch	Number of branches of a GitHub project.
	Releases	Number of releases of a Github project.
	NbreFiles	Number of files within a GitHub project.
	NbreAuthors	Number of authors of a GitHub project.
	NbreCommits	Total number of commits in a GitHub project.

repeated across all projects using a given framework, this might introduce bias towards the most common configuration frameworks, yielding models that are overfitted on System and Properties. To counter this, we resampled the training set to obtain the same number of instances for each configuration framework. Based on the number of non-fork projects in Dataset 2, we chose 30 projects per framework, which required us to resample jConfig, Constretto and CFG4J with replacement (i.e., duplicating some of their rows) to obtain 30 data points. The resulting resampled data set forms our training set.

We used VIF (Variance Inflation Factor) analysis to remove highly correlated variables (VIF > 5 [15]), then built an initial baseline model only containing the project-related variables as independent variables. We then incrementally add the three configuration framework dimensions of the taxonomy to evaluate the degree to which each dimension adds new information related to the maintenance effort-related dependent variable (we build separate models for each of the three maintenance effort measures). We use ANOVA tests and AIC score to evaluate how significantly a new model differs to an earlier model. We then calculate precision, recall and AUC values (via 10-fold cross-validation) to evaluate how well the models fit the data in terms of lack of false alarms, ability to find all known high maintenance projects and performance improvement compared to a random model, respectively. We also build a final model with only the statistically significant variables.

Finally, to understand which variables have the highest impact on the dependent variable as well as the direction (increasing/decreasing) of this impact, we used the effect size score of Shihab et al. [29]. It requires evaluating a classification model using the median value of each variable as input, then, one at a time, adding one standard deviation to the median value of a variable (while keeping the other attributes at their median value). For example, if the model output is 50% when all independent variables are at their median value, while the output after adding a standard deviation to the first independent variable is 100%, we say that the latter independent variable has an effect score of $\frac{100\%-50\%}{50\%} = 1$, indicating a 100% increase in probability compared to the baseline effect size. Whereas one cannot directly compare the coefficients of the logistic regression models, as not all variables use the same unit [15], the effect sizes of each variable can be directly compared to each other.

TABLE VI

MODEL FOR RQ3 (AIC: 355.63, PREC.: 77.41%, RECALL: 72.72%).

Attribute	Coefficient	Std. Error	Signif. code	Effect size
NbreAuthors	0.009478	0.002097	***	1.72494E-05
Releases	0.001029	0.0002324	***	2.07543E-07
NbreCommits	-0.0002525	0.00005047	***	-1.10599E-08
ActMaint	1.679	0.3764	***	2.400912759
PersisVarietyLow	1.214	0.34	***	1.564812448
AgeLow	1.167	0.3019	***	1.485730827
QualDocMed	-1.277	0.506	*	-0.691757625

For boolean variables, we used "false" as the reference value for the effect size, and used the value "true" instead of the "median value + standard deviation". Hence, the effect size expresses the effect of moving from "false" to "true".

RQ3. Which factors impact the percentage of configuration-related commits?

Finding 9: The taxonomy variables have a higher impact on maintenance effort in terms of number of commits.

Table VI shows the significant variables in the logistic regression model, split up in project-related and configuration framework-related. In both groups, variables are ordered based on the absolute value of the effect size, from highest effect (either positive or negative) to lowest. It is easy to see how the three project-related variables have only a tiny effect size, close to zero, while the taxonomy-related variables have an effect size of at least 69.1% (negative; QualDocMed).

We find that **the more active the developers of the configuration framework are (ActMaint), less choice of storage formats for configuration files (PersisVarietyLow), or younger configuration frameworks (AgeLow), the more maintenance effort** the client developers of the framework seem to perform. On the other hand, a **lower quality of documentation (medium as opposed to high; QualDocMed) seem to be linked to lower effort**.

Surprisingly, we found that the higher the quality of a configuration framework's documentation, the more commits the configuration framework requires. This could be explained by the fact that frameworks with *more comprehensive* documentation might be more feature-rich, or that developers look for easy-to-use configuration frameworks (Section V), which tend to have concise documentation. More research is needed to evaluate the needs for good configuration framework documentation.

TABLE VII

MODEL FOR RQ4 (AIC: 331.87, PREC.: 74.5%, RECALL: 90.30%).

Attribute	Coefficient	Std. Error	Signif. code	Effect size
Releases	-0.003724	0.002345		-2.90491E-08
NbreFiles	-0.0006832	0.0001462	***	-3.32257E-10
NbreCommits	0.00007836	0.00003293	*	8.5836E-12
ScopeUniv	-3.797	0.7478	***	-0.12657923
QualDocMed	-2.841	0.8845	**	-0.050931693
AgeMed	-2.203	0.489	***	-0.026086136
ActMaint	-1.825	0.4438	***	-0.017012309
HierStruct	0.8339	0.367	*	0.001885136

TABLE VIII

MODEL FOR RQ5 (AIC: 388.69, PREC. : 64.64%, RECALL: 70.90%).

Attribute	Coefficient	Std. Error	Signif. code	Effect size
Branch	-0.005857	0.004467		-1.71518E-05
PersisVarietyLow	2.235964	0.468840	***	1.412526089
ActMaint	1.473443	0.374606	***	1.021655129
JDKStandard	-1.896858	0.449827	***	-0.78785475
QualDocMed	-1.599531	0.514350	**	-0.721445889
HierStruct	-0.784968	0.322907	*	-0.438725736

RQ4. Which factors impact the percentage of configuration-related source files?

Finding 10: Taxonomy variables again have the strongest link with maintenance effort.

We obtain similar findings as for RQ3 (see Table VII), with ActMaint, AgeMed (replacing AgeLow) and QualDocMed again figuring amongst the most highly impacting variables, although the sign of the effect size has changed. Furthermore, ScopeUniv and HierStruct are additional impactful variables.

We find that **younger (AgeMed, as compared to Age-High) and actively maintained (ActMaint) configuration frameworks, with less documentation (QualDocMed) and a universal scope (ScopeUniv), are spread across less files** in a given source project. Frameworks that support **hierarchical configuration models (HierStruct) are spread across more files**, although this effect size is close to zero.

The finding for AgeMed is as expected, since older frameworks have been used longer by projects, leading to tighter integration. Moreover, this finding can give an indication that developers progressively add configuration frameworks in different source code files, which leads to stronger coupling to the configuration framework.

The scope of configuration frameworks is the third most important factor, indicating that **less general-purpose frameworks like Spring have a stronger coupling** to a software system, likely because they come with more heavy configuration (and other) machinery. The HierStruct finding seems to suggest that hierarchical configuration models have a stronger coupling with source code projects, i.e., require more complicated interactions in a project.

RQ5. Which factors impact the percentage of developers touching configuration code?

Finding 11: Taxonomy variables have the highest impact.

Since RQ5 yields **similar results as RQ3** (Table VIII), we only discuss the differences. The age of the configuration frameworks does not play a major role, and is replaced by the variables JDKStandard and HierStruct. In particular, **Java SDK frameworks have less developers making changes** to configuration-related code (likely due to their simplicity), while the use of **hierarchical configuration models sees less developers do changes** (in more files, see RQ4).

B. Discussion

Based on the three explanatory models, we conclude that configuration framework taxonomy metrics have important relations with the effort required to maintain configuration frameworks in a given software project. Surprisingly, none of the significant configuration framework variables belongs to the Programming Support dimension (Table II), while all 5 General Properties and 2 out of 7 Feature Richness properties were significant in at least one model.

One of the two most commonly impactful General Properties is the degree of active development of a configuration framework. The more active, the more commits and developers need to be involved to maintain a client project, although slightly less files in the project actually use the framework. This basically empirically confirms the common knowledge that code reuse implies staying on the lookout for updates.

The other commonly impactful General Property is the quality of documentation. Higher quality involves more maintenance commits by more developers across more files. This was the most surprising finding. We believe that a correct interpretation of this finding is that documentation should be *"simple but not simplistic"*. It should emphasize concise examples about how to integrate a configuration framework, as developers generally have a lack of time to deeply learn a configuration framework.

Older, stable frameworks require less maintenance commits (likely because the framework is less active), but their usage typically is spread across more files in a project. Finally, hierarchical configuration models were associated with less developers making commits, but those commits touched more files of a project.

VII. THREATS TO VALIDITY

Regarding threats to external validity, we only considered general-purpose configuration frameworks for Java projects. However, according to Nagappan et al.'s diversity measures [18] (value close to 1), our sample of repositories is representative for the population of Java projects in GitHub. In future work, we plan to investigate other programming languages, as well as domain-specific configuration frameworks like SharedPreferences.

Regarding construct validity, we use the percentage of changes, files or authors touching configuration framework usage as proxies for maintenance effort. Although these are typical approximations for maintenance effort case studies [34], they are still proxies. Although we studied three different proxies, using other metrics should be considered in future work. Note that we did not distinguish bug fix commits from commits adding new features or using new framework APIs, as those would require additional heuristics for analyzing our data on top of the ones we use to identify configuration commits. However, our models do control for

the total activity in a project, hence we believe this threat is addressed in a reasonable manner.

Finally, regarding internal validity, we conducted a short survey for which we received 10 answers. We plan to conduct a larger survey to investigate configuration frameworks and their impact on configuration errors and maintenance effort.

VIII. Conclusions

We conducted an empirical study on 11 major Java configuration frameworks in almost 2,000 GitHub projects. A proposed taxonomy of framework features provides evidence of the wide variety of features offered by configuration frameworks. It also supports practitioners in selecting frameworks whose features are most suitable to the projects.

We found that SDK frameworks are the most commonly used frameworks, in half of the cases complemented by more powerful frameworks. Simple frameworks are typically preferred. Also, the choice of configuration framework explains to a large degree the maintenance effort required for configuration, whereas project-specific characteristics play less of a role. More active frameworks with higher quality documentation that have more flexible configuration storage formats have a higher tendency towards more configuration-related changes by more developers. On the other hand, older, less active configuration frameworks with high documentation tend to be used across more files in a software project.

Our findings empirically confirm common assumptions about configuration frameworks (e.g., reuse requires more maintenance if reused framework is active). However, they also raise new questions about the role of configuration framework documentation or the perils of hierarchical configuration models (stronger coupling), which should be addressed in future work.

References

[1] Apache commons configuration. https://commons.apache.org/proper/commons-configuration/.

[2] F. A. Arshad, R. J. Krause, and S. Bagchi. Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 198–207, Nov 2013.

[3] Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. Users beware: Preference inconsistencies ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, USA, 2015. ACM.

[4] Carbon. http://carbon.sourceforge.net/WhatIsCarbon.html.

[5] Cfg4j. http://www.cfg4j.org/.

[6] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter. Prevalence and maintenance of automated functional tests for web applications. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 141–150, Sept 2014.

[7] Constretto. http://constretto.org/.

[8] Deltaspike. https://deltaspike.apache.org/.

[9] Victor Denisov. Overview of java application configuration frameworks. *Inter Journal of Open Information Technologies*, 1(6), 2013.

[10] Zhen Dong, Artur Andrzejak, and Kun Shao. Practical and accurate pinpointing of configuration errors using static analysis. In *Proc. of Inter. Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015.

[11] Zhen Dong, Mohammadreza Ghanavati, and Artur Andrzejak. Automated diagnosis of software misconfigurations based on static analysis. In *Inter. Workshop of Program Debugging (IWPD) at ISSRE 2013*.

[12] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12. IEEE Computer Society, 1986.

[13] jconfig. https://sourceforge.net/projects/jconfig/.

[14] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 215–224, 2014.

[15] Robert Kabacoff. *R in Action: Data Analysis and Graphics with R*. Manning Publications Co., Greenwich, CT, USA, 2015.

[16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, 2014.

[17] A. Kapoor. Web-to-host: Reducing total cost of ownership. Technical report, Technical Report 200503, The Tolly Group, May 2000.

[18] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in Software Engineering Research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA, 2013. ACM.

[19] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI'04*, Berkeley, CA, USA, 2004.

[20] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, 2003.

[21] Owner. http://owner.aeonbits.org/.

[22] Play framework. https://www.playframework.com/.

[23] A. Rabkin and R.H. Katz. How hadoop clusters break. *Software, IEEE*, 30(4):88–94, July 2013.

[24] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *Proc. of the 2011 26th IEEE/ACM Intl. Conf. on Automated Software Engineering*, ASE '11, pages 193–202, 2011.

[25] Raihan's jconfig. https://github.com/prshreshtha/jconfig.

[26] Mohammed Sayagh and Bram Adams. Multi-layer software configuration – empirical study on wordpress. In *Proc. of the 15th IEEE Inter. Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 31–40, Bremen, Germany, September 2015.

[27] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. On cross-stack configuration errors. In *Proc. of the 39th Inter. Conference on Software Engineering (ICSE)*, Buenos Aires, Argentina, May 2017.

[28] SharedPreferences. https://developer.android.com/reference/android/content/SharedPreferences.html.

[29] Emad Shihab, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, 55(11):1981–1993, 2013.

[30] Spring framework. https://projects.spring.io/spring-framework/.

[31] Developer's survey via google forms. https://goo.gl/kXCQ7Q.

[32] Anatole Tresch. Java configuration. http://javaeeconfig.blogspot.de/2014/08/overview-of-existing-configuration.html.

[33] Typesafe. https://github.com/typesafehub/config.

[34] Hong Wu, Lin Shi, Celia Chen, Qing Wang, and Barry W. Boehm. Maintenance effort estimation for open source software: A systematic literature review. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 32–43, 2016.

[35] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, USA. ACM.

[36] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[37] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, 2011.

[38] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *Proc. of the 34th International Conference on Software Engineering*, San Francisco, CA, USA, May 22–24, 2013.

[39] Sai Zhang and Michael D. Ernst. Which Configuration Option Should I Change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 152–163, New York, NY, USA, 2014. ACM.