Noname manuscript No.
(will be inserted by the editor)

What should your Run-time Configuration Framework do to Help Developers?

Mohammed Sayagh · Noureddine Kerzazi · Fabio Petrillo · Khalil Bennani · Bram Adams

Received: date / Accepted: date

Abstract The users or deployment engineers of a software system can adapt such a system to a wide range of deployment and usage scenarios by changing the value of configuration options, for example by disabling unnecessary features, tweaking performance-related parameters or specifying GUI preferences. However, the literature agrees that the flexibility of such options comes at a price: misconfigured options can lead a software system to crash in the production environment, while even in the absence of such configuration errors, a large number of configuration options makes a software system more complicated to deploy and use. In earlier work, we also found that developers who intend to make their application configurable face 22 challenges that impact their configuration engineering activities, ranging from technical to management-related or even inherent to the domain of configuration engineering.

In this paper, we use a prototyping approach to derive and empirically evaluate requirements for tool support able to deal with 13 (primarily technical) configuration engineering challenges. In particular, via a set of interviews with domain experts, we identify 4 requirements by soliciting feedback on an incrementally evolving prototype. The resulting "Config2Code" prototype, which implements the 4 requirements, is then empirically evaluated via a user study

Mohammed Sayagh
E-mail: msayagh@cs.queensu.ca
Queen's University, Canada.

Noureddine Kerzazi
E-mail: n.kerzazi@um5s.net.ma
ENSIAS, Mohammed V University, Morocco.

Fabio Petrillo
E-mail: fabio.petrillo@uqac.ca
Université du Québec à Chicoutimi, Canada.

Khalil Bennani, Bram Adams
E-mail: {khalil.bennani, bram.adams}@polymtl.ca
Polytechnique Montreal, Canada.

involving 55 participants that comprises 10 typical configuration engineering tasks, ranging from the creation, comprehension, refactoring, and reviewing of configuration options to the quality assurance of options and debugging of configuration failures. A configuration framework satisfying the 4 requirements enables developers to perform more accurately and more swiftly in 70% and 60% (respectively) of the configuration engineering tasks than a state-of-the-practice framework not satisfying the requirements. Furthermore, such a framework allows to reduce the time taken for these tasks by up to 94.62%, being slower for only one task.

Keywords run-time software configuration · configuration engineering · empirical study · user study .

1 Introduction

Successful software applications need to be adapted to different usage scenarios. To achieve this, applications rely on "configuration options whose value can still be changed by the end user, without having to re-deploy" [1]. Hence, users can customize the behaviour of an application depending on their own preferences (e.g., GUI preferences or account information), enable or disable the features that are important to them, or tune the performance of the software application without any modification or recompilation of the source code. As such, each decision that depends on the user's usage scenarios should be postponed until the execution of the system, resulting in potentially hundreds (e.g., Apache Hadoop) or even thousands (e.g., Mozilla Firefox) of options in a given application.

Concretely, when a developer introduces a new feature or wants to make an existing behaviour configurable, she first needs to create a new configuration option with a succinct name (e.g., `enable-undo`) and adequate type (e.g., `boolean`), specify any other constraints for the option (e.g., `enable-undo` implies `enable-redo`), select a storage medium for the option (e.g., configuration file), and specify a default value for the option in that medium (e.g., `enable-undo=true`) [1]. Before releasing the feature to the user, the developer still needs to test the application with the option's possible values (e.g., "true" and "false" for a boolean option), update the user documentation to include the new option and its metadata, and possibly update the developers' internal wiki about the expected lifetime of the option (e.g., permanent feature vs. temporary feature toggle [2]).

Although these activities are a part of the established process of run-time configuration engineering followed by practitioners, many of these activities lack tool support and/or research initiatives, as identified in our prior work through 14 interviews, a survey with 229 respondents, and a systematic literature review [1]. Specifically, while configuration activities like debugging of configuration errors [3,4,5], choice of default values [6,7] and configuration-aware testing [8,9,10] have been studied in depth, the other activities are not. For example, while the list of accessible configuration options at all times needs

to remain synchronized with the options that are actually used in the source code [11], we found that developers often do not clean up unused options because they are afraid of accidentally removing essential functionality [1]. Similar to source code, documentation of options often is outdated, while the impact of an option on the code is hard to understand manually, since developers not always use consistent naming conventions or even configuration access APIs.

During our prior interviews [1], the interviewed developers, architects and managers also provided several requirements for run-time configuration framework support that (in their eyes) could resolve some of the challenges that they were facing. Since these requirements initially were high-level, incomplete or vague, we decided to use a rapid prototyping process [12] aimed at fleshing out the core requirements of an effective framework for run-time software configuration. Basically, at the end of each interview, we would show the prototype available at that point in time, obtain feedback, then iterate over the design to produce a new prototype for the next interview. Apart from helping to derive core tool requirements, the resulting prototype also allowed us to empirically compare the impact of these requirements to a state-of-the-practice configuration framework. This empirical evaluation was done through a user study on 10 configuration engineering tasks derived from the configuration engineering process discussed above, featuring 55 participants from both industry and academia, and spread across two continents.

Our main contributions in this work can be listed as follows:

- Identification of 4 major requirements for a run-time configuration engineering framework aimed at improving a software system's configuration in terms of its comprehensibility, correctness, and maintainability.
- Large user study with 55 participants (industry/academia) and empirical analysis of the impact of the 4 requirements on 10 typical configuration tasks.
- A prototype implementation of a configuration framework ("Config2Code") that implements the 4 requirements [13].

2 Background and Related Work

2.1 Software Configuration

Software configuration allows postponing a decision in a software system until the required information is available, typically at deployment- or run-time [12]. These decisions can be made explicit in the user requirements [15], emerge during development to anticipate use cases of advanced users, be related to testing (enabling/disabling features on the fly), or pop up during the evolution of the code base. For example, agile developers often focus on getting the logic right for a specific use case without being disturbed about generalization ("You Ain't Gonna Need It" principle). Later on, during refactoring, the functionality can be generalized by "externalizing" hard coded numbers and string literals

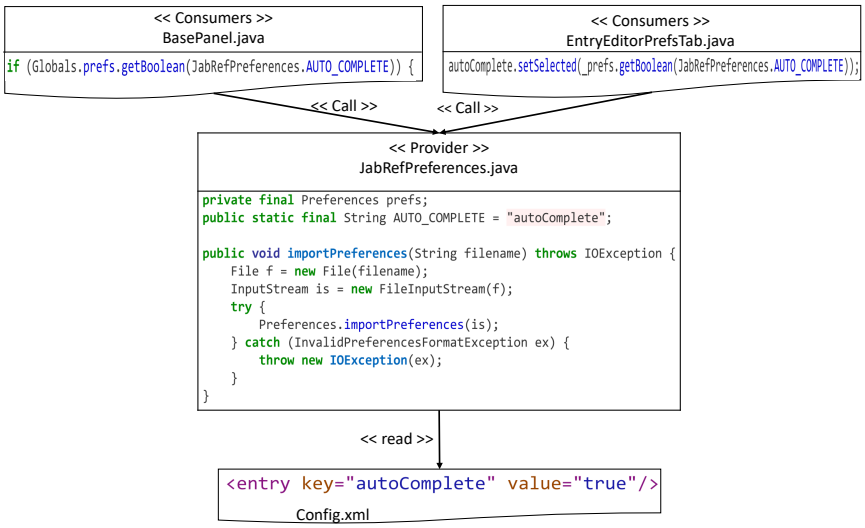

Fig. 1 Code example from the configuration system of the JabRef open source project [14]. The provider class `JabRefPreferences` reads user choices from the `Config.xml` configuration file into the attribute `prefs` (using the *Preferences* framework), while the `BasePanel` and `EntryEditorPrefsTab` consumer classes access this attribute to use the corresponding configuration option.

into variables ("configuration options") whose value is loaded from some kind of configuration storage medium, typically a file, database or the application's program arguments. Options can range from host names to tuning parameters, debugging options and feature toggles [16].

Figure 1 shows a concrete example of the configuration system of the JabRef open source project. It consists of an XML configuration file (storage medium) that has more than 150 configuration options, including the **''autoComplete''** configuration option highlighted in Figure 1. That option can be switched on or off by end-users to enable or disable the auto complete feature. JabRef's *JabRefPreferences* class is a so-called provider class responsible for loading these configuration options in the attribute *prefs*. This class then provides configuration values to configuration "consumer" classes of *JabRef* such as `BasePanel` and `EntryEditorPrefsTab`. As shown in Figure 1, *JabRef* uses the configuration framework *Preferences*. While many frameworks dedicated to configuration exist, we have found in our prior work on the usage of configuration frameworks on Github projects that 46% of our studied projects still roll their own configuration class [17].

Different types of configuration options exist, yet this paper focuses on "run-time" configuration options. Run-time configuration options are "configuration options whose value can still be changed by the end user, without having to re-deploy" [1], for example by overriding the options' default value

via the command line or by changing the value in a configuration file. Such options require an application to actively check their value upon start-up or at specific times during execution.

In contrast, other types of configuration options exist that limit configurability to earlier stages in the lifetime of an application, such as its compilation or deployment. Such configuration decisions cannot be revoked or overridden afterwards by the end user. For example, compile-time configuration options usually select the code snippets (via conditional compilation) or entire files (via build scripts) that will be considered by the compiler. Similarly, deploying a compiled software application requires configuring the execution environment [18] of the application, i.e., physical or virtual machines (containers) with the right operating system, database, web server and 3rd party libraries. For this kind of configuration, organizations typically use domain-specific languages (DSLs) such as Puppet, Chef or Ansible, typically referred to as "infrastructure-as-code" (IaC) [16].

2.2 Related Work

As discussed above, there is a tight relation between run-time configuration and the popular concept of Infrastructure-as-Code (IaC) [16]. Both infrastructure and software development teams build and manage infrastructure using automated "Infrastructure as Code" (IaC) tools [19]. The infrastructure team is responsible for, amongst others, automatically building the environment in which an application should be deployed [16]. In contrast, the development team traditionally has been responsible for developing highly-configurable software applications [12]. The research and practices related to both teams, while evolving separately for a long time, currently are converging under the influence of DevOps [18]. This is an evolution of agile development that focuses on bringing developers and operators together in order to synchronize development and production [20].

This synergy between code and infrastructure has led to massive adoption of IaC by modern software organizations, in the form of declarative specifications in a domain-specific language like Puppet [21] and Ansible [22]. These specifications effectively are a form of source code, whose "compilation" in this case generates the desired environment (e.g., virtual machine). Researchers have indeed identified several source code phenomena in IaC code. For example, Jiang et al. [23] studied the co-evolution of Puppet and Chef configuration files with source code, tests, and builds, and found a tight coupling of IaC file changes with test files. Sharma et al. [24] empirically studied a catalog of 13 implementation and 11 design configuration smells. The design configuration smells were shown to have 9% higher average co-occurrence among themselves than the implementation configuration smells, suggesting that the developers should pay more attention to the former. In contrast to the work on IaC, we focus on the synergy between the *run-time* configuration options of a software system and its code base.

Another line of related work primarily focuses on dealing with run-time configuration options and their related problems. We provide a systematic literature survey elsewhere [1], and instead focus here on the most closely related work. Zhang et al. [6] addressed the concern of finding the right option to be changed in order to obtain an application's desired behavior. They introduced a technique based on dynamic profiling and static analysis, supported by a tool called *ConfSuggester* to help debug configuration errors. Related to this, Huang et al. proposed *ConfValley* [25], a declarative language to express and check configuration specifications. Li et al. presented a tool called *ConfTest* [26] to prevent misconfiguration, and evaluated it against injected misconfigurations. Dong et al. proposed an approach called *ORPLocator* [27] to support detection of inconsistencies between source code and documentation via static analysis. For each configuration option, they identify the source code locations reading it, then compare the results against the option names listed in the documentation. Similarly, Jin et al. presented *PrefFinder* [28], an NLP framework that navigates scarce, distributed documentation to find inconsistencies.

Our study is fundamentally different from the above work on software configuration. Instead of focusing specifically on debugging of configuration errors or finding the best default value of an option, we identify and empirically validate 4 core requirements of configuration engineering able to address a wide range of configuration challenges by easing their management in order to improve software configuration quality in terms of comprehensibility, maintainability, and correctness.

A final line of related work on software configuration focuses on product line engineering [29,30,31]. A product line is a set of practices and architectures that allow to build a line of related software products ("variants") around a common platform. Developers can extend the platform with features between which certain constraints can exist (e.g., feature A requires feature B). Variants can then be produced by enabling subsets of the features, while respecting their constraints. A large body of research exists on product lines, so here we focus on product line research related to the configuration of the features going into variants. For example, Hubaux et al. [32] found that Linux and eCos users, two large operating system product lines, face a lack of documentation on how to configure these software systems. Nadi et al. [33] proposed an approach that identifies the feature constraints of C-based product lines, such as Linux, from the source code. Medeiros et al. [34] compared ten sampling algorithms to identify which configurations to test in a product line software system.

Our study on run-time configuration options is complementary to existing research on product line engineering. For example, in order to build variants, product line features can be represented as configuration options, between which some constraints can be defined, that need to be configured. This is similar as to how run-time configuration options can be used to configure the features a user wants to enable during execution. However, run-time configuration options do not necessarily map one-to-one to features, but can customize any aspect of the behaviour of a software system, for example to tweak its

activity	challenge
1. creation of options	ad hoc planning of options (M) adding options increases complexity (I) choosing widely applicable default value (I) unclear configuration ownership (M)
2. managing storage media	**mixing media increases complexity** (T) choice of media impacts performance (T)
3. managing option type	**choice of option type** (T) configuration variants across environments (T)
4. configuration access in code	**coupling between ProviderClass and ConsumerClass** (T) adoption of dedicated frameworks (M)
5. comprehension of options	**unknown impact of option (change)** (T) **lack of option comprehension tools** (T) **meaningless option names** (T)
6. maintenance of options	**option removal is risky** (T) **traceability between options and code** (T)
7. resolving configuration failures	**debugging config. failures is hard** (T) **lack of configuration debugging tools** (T) no strategy for avoiding config. regression (M)
8. knowledge sharing	**lack of option documentation** (M) no internal communication about options (M)
9. quality assurance	**code review ignores configuration** (I) **lack of automatic config. validation** (T)

Table 1 Overview of challenges related to configuration activities [1], which are either (M)anagement-related, (I)nherent or (T)echnical. The framework requirements identified in this paper focus on the challenges in bold.

execution performance, specify file paths and other locations of resources, or even to customize the GUI of the system. In that sense, our work differs from the domain of product line engineering.

3 Challenges of Run-time Software Configuration Engineering

While run-time configuration has been an ongoing concern in software development for decades [12], practitioners still suffer from a wide range of challenges involved with it. In particular, in order to provide and maintain a typical configuration system as shown in Figure 1, an organization needs to implement a configuration engineering process, i.e., a "discipline that encompasses activities involved in the creation, integration, and maintenance of run-time configuration options in a software application" [1]. For example, when a certain functionality should become configurable, one needs to add a new configuration option, pick a good name for it, then access the option's value within the code to decide when the configurable code should be enabled, or to tweak the code's behaviour in some other way.

In earlier work [1], we performed interviews with 14 experts, a large survey with 229 software engineers and a systematic literature review to recover

and understand the typical configuration engineering process followed in practice, as well as to identify challenges and potential solutions. We distilled 9 major configuration activities and 22 related challenges, as summarized in Table 1. For each challenge, the table also indicates whether it is related to (M)anagement choices, (T)echnical difficulties, or (I)nherent difficulties of software configuration. A framework most likely would only be able to deal with the latter two kinds of challenges.

3.1 The (M)anagement-related Challenges

The (M)anagement-related challenges involve the need for explicit planning of options within the development process (e.g., based on requirements), clear assignment for each option of an "owner" responsible for coordinating code changes, explicit evaluation and adoption of the right framework (library) for managing and accessing configuration options, a strategy to avoid regressions of configuration failures (e.g., by forcing developers to document incorrect option values in a wiki), enforcing guidelines for documenting options for end users and clear communication amongst developers regarding the goal and impact of options. Most of these challenges require process-level changes and follow-up, while some, such as the lack of option documentation [35], could also benefit from better technical support.

3.2 The (T)echnical challenges

The (T)echnical challenges refer to implementation-related challenges, although they require more than just the selection and adoption of a dedicated configuration framework (which in itself is a challenge as well [17]). For example, substantial technical support is needed to determine the impact of an option on different parts of the code base [10], to choose meaningful option names (enabling easy understanding) and to automatically validate constraints on the value of configuration options [8].

3.3 The (I)nherent Challenges

In contrast, the challenges (I)nherent to software configuration cannot be avoided through better organization or tooling; one can only try to reduce their impact. For example, any added option increases the list of options to read and understand [6], possibly discouraging or at least puzzling potential users [5]. Similarly, the default value of an option has to be chosen in such a way to enable plug-and-play functionality for most of the end users, which is surprisingly hard to achieve. The unclear link between configuration options and the source code impacted by it [36], as well as the focus of code review on changed source code lines only, even make code review of configuration-related changes a challenge.

3.4 The Challenges Considered by this Paper

Instead of focusing on one particular challenge, this paper identifies and empirically evaluates four basic requirements obtained throughout our earlier interviews and questionnaire [1] that have the potential of resolving the technical and inherent challenges as shown in bold in Figure 1. We consider these challenges in particular since they can be addressed by an automatic approach or a framework, while the other challenges can only be addressed by management decisions (left for future work). In particular, we address the following challenges (we refer to our prior work [1] for more details about each of these challenges):

Mixing Storage Media Increases Complexity: A large number of storage media makes the configuration of a software system as well as the maintenance of a software configuration option challenging. We found in our prior work that due to the weak communication between developers and because each developer has her own preferences for storing configuration options, developers can end up with a large number of different storage media to store configuration options (e.g., mixture of XML, INI and JSON files), possibly spread across different folders. Therefore, finding the right medium and option to change becomes more challenging.

Choice of Option Type: 10 respondents to our survey faced slow-down problems due to the incorrect data format they choose for their configuration option, while other respondents faced problems related to how to express the type of a configuration option (e.g., an option that has other sub-options).

Coupling between ProviderClass and ConsumerClass: Accessing and reading the value of options from different source code locations makes understanding, maintaining, and debugging configuration options challenging [1]. We found in our prior work that 40% of surveyed developers read and use configuration options in different classes without defining a clear API dedicated to the configuration options.

Unknown Impact of Option (change): Because developers can read and use a configuration option from different locations of the source code (previous challenge), knowing the impact of changing a configuration option is not trivial, as developers cannot know in advance the impact of that change. In fact, we found in our prior work [1] that only 31% of surveyed developers know the impact of all configuration options.

Lack of Option Comprehension Tools: Understanding the goal of configuration options is challenging due to the low quality of configuration options documentation. That was confirmed by one surveyed developer's experience: *"The one [who] created that option has quit the team and that option [is] invoked in too many places of the code and hard to guess what it does"* [1].

Meaningless Option Names: We also found in our prior work that configuration options generally do not have meaningful names and do not respect a naming convention. We have found that only 54% of surveyed developers follow a predefined naming convention for configuration options.

Option Removal is Risky: As developers do not know the impact of a configuration change, we found that refactoring options and even cleaning dead or unused options is risky. In fact, dead options are left in the configuration files as confirmed by an interviewed expert: *"... we don't clean the configuration files, because we don't know when and where the system can crash. Dead options are kept in the configuration files forever"* [1].

Traceability between Options and Code: Surprisingly, we found that developers do not keep track of configuration option changes in a version control repository. This is because many developers do not consider run-time configuration files to be code, but instead consider them as external artefacts. This leads to losing the traceability of configuration options.

Debugging Configuration Failures is Hard: Debugging configuration errors is not straightforward, particularly when a misconfigured software system has a large number of options. Therefore, we found in our prior work [1] that developers use the same mechanisms to debug configuration errors as any other kind of bug.

Lack of Configuration Debugging Tools: As discussed before, developers (ab)use ordinary debugging approaches to also debug configuration errors. They do not tend to use the existing techniques in the literature for debugging configuration errors.

Lack of Option Documentation: Documentation of options includes adding comments to configuration files and creating a clear documentation in a Wiki or web-page. We found that this is mostly ignored by developers.

Code Review Ignores Configuration: As stated earlier, developers do not consider run-time configuration to be code and, hence, we found that 36% of surveyed developers do not review configuration files.

Lack of Automatic Validation: Many of the interviewed and surveyed developers do not validate the correctness of user configuration choices in the source code. Therefore, users can assign incorrect or unexpected values to configuration options that might lead to incorrect behaviour. Ideally, developers should verify at application load-time the correctness of its configuration values.

4 Core Requirements for Run-time Configuration Frameworks

This section discusses the prototyping approach used to identify the core requirements for run-time configuration framework support, followed by a detailed discussion of each of the four identified requirements.

4.1 Identification of Requirements

The major trigger for this work occurred during one of our initial interviews about the run-time configuration engineering process [1], when one of the industrial interviewees exclaimed that "*Run-time configuration is code too*".

While this principle sounds very similar to Infrastructure-as-Code (IaC) [16] (see subsection 2.2), the interviewee actually meant something different: conceptually, run-time configuration options and metadata like types, constraints and default values are tightly coupled with the source code and its development process, while in practice they are physically separated, without explicit links. If, somehow, configuration options and their metadata could become part of the source code, and automatically co-evolve with it, developers could leverage traditional tools for testing, refactoring and debugging source code to perform maintenance, comprehension tasks and quality assurance of configuration options.

Since the extent of this principle and its impact on run-time configuration engineering activities still was rather vague to derive concrete requirements for, we decided to adopt a rapid prototyping process [12]. Such a process typically is used when the requirements of a software product are unclear. The end goal of such a process is not the prototype itself, but rather a specific list of requirements that can then be used to implement a real product (e.g., using an iterative process). Similarly, our aim was to obtain a list of core requirements for run-time configuration frameworks able to resolve a wide range of configuration challenges (see Table 1). In addition, the prototype would serve as vehicle for a user study that empirically evaluates the impact of the requirements on the ability to perform the typical configuration engineering activities.

Our rapid prototyping process basically consisted of the following activities:

1. extract new ideas proposed by an interviewee from the interview transcript
2. derive requirements from these ideas
3. compare and integrate these requirements with those already prototyped
4. adapt the prototype's existing features and add new ones according to the new list of requirements
5. test and stabilize the prototype on example code snippets
6. at the end of the next interview, solicit new ideas for tool support from the interviewee
7. once all ideas have been noted down, demo the current prototype to the interviewee
8. obtain and record feedback about the prototype and its relation to the interviewee's own ideas
9. iterate back to step 1

The choice of this process instead of, for example, a design science methodology [37] or another requirements soliciting methodology was largely motivated by the limited availability of interviewees during our study. We basically only had one shot with them, hence repeated one-on-one feedback during prolonged periods was impossible. For that reason, we opted for longitudinal feedback across 12 interviews and interviewees (we started the initial prototype after the second interview). Given that we aimed to find the major requirements for tool support, by definition those requirements should be shared by the majority of these 12 interviewees.

Table 2 Interviewed subjects' experience and role [1].

Subject	Experience (#years)	Role
P1	28	Developer and Manager
P2	27	Researcher
P3	21	Developer
P4	15	Manager
P5	14	Developer and Industrial Researcher
P6	14	Infrastructure Architect
P7	13	Manager
P8	12	Developer and Architect
P9	10	Developer and Architect
P10	10	Developer
P11	8	Integrator
P12	7	Developer

The first two authors conducted the face-to-face interviews with 12 senior developers and team leads, who belong to 11 different companies located in four countries and who are working in large organizations (dozens to a few hundreds of developers). Table 2 shows the experience and role of these interviewees. We interviewed senior developers and team leaders since they had more experience in software development and dealing with configuration issues than junior developers, and they likely met more problems in their respective context. Each semi-structured interview lasted more than 60 minutes and was divided into three parts. In the first part, we asked questions related to the key practices for managing configuration options and the roles responsible for adding, editing, and maintaining configuration options. In the second part, we asked open-ended questions about typical configuration issues, and the life cycle of options from development to deployment into the production environment. In the third part, we applied the rapid prototyping process discussed above.

Once the final prototype was obtained, we extracted the list of features of the final prototype, then used open coding to group related features into more abstract requirements. Open coding [38] allowed us to achieve this by tagging features with codes, then to abstract up from the codes by grouping related codes into higher-level categories. We repeated the process until the codes and categories were saturated. The resulting categories then correspond to core requirements for configuration frameworks. Apart from a number of utility and convenience features/requirements, we ended up with 4 core requirements that we think cover the 13 challenges, as shown in Table 3.

4.2 The Four Requirements

This subsection discusses the 4 requirements, while the next section presents the implementation details of the resulting prototype from which the requirements were extracted.

Challenge	Task	R1	R2	R3	R4
mixing configuration media increases complexity				+	
choice of option type	1	+			
coupling between Provider- and ConsumerClass	1,3		+		
unknown impact of option (change)	3	+	+		
lack of option comprehension tools	4	+	+	+	+
meaningless option names	6				+
option removal is risky	3	+	+	+	
traceability between options and code	1,2,3	+		+	
debugging configuration failures is hard	5	+			
lack of configuration debugging tools	5	+			
lack of option documentation	1			+	+
code review ignores configuration	7	+	+		
lack of automatic configuration validation	6				+

Table 3 Mapping the requirements R1 (Options-as-Code), R2 (Encapsulation of Configuration Access), R3 (Generation of Configuration Media) and R4 (Automatic Configuration Validation) to the bold challenges of Table 1 and the 7 tasks evaluated in our user study: Task 1 (creation of configuration options), Task 2 (refactoring - changing a default value), Task 3 (refactoring - removing configuration options), Task 4 (comprehension of options), Task 5 (fixing a configuration error), Task 6 (configuration quality), and Task 7 (configuration review).

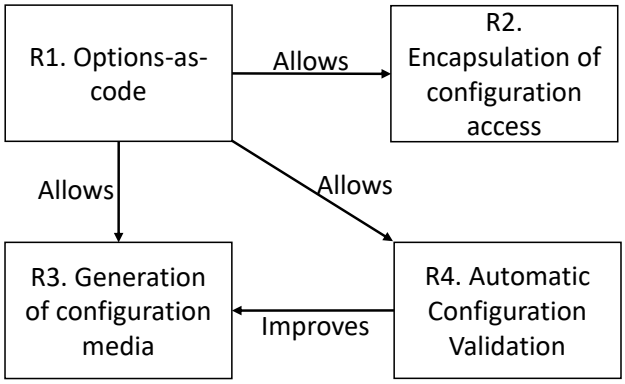

Fig. 2 The relation between the four principles.

R1. Options-as-Code

The first requirement states that **a configuration framework should physically integrate the definition of a configuration option and its related metadata (e.g., type, default value, description and constraints) into the source code**, for example into the ProviderClass in Figure 1. This requirement not only allows developers to remain inside the source code to create a new configuration option, but by adopting a specific syntax or idiom to specify options it is straightforward for frameworks or manual search queries to identify all options of an application or to serve as starting point for refactoring [39].

```
public class JabRefPreferences {
  @Config(name = "autoComplete",
    namespace = "papers.search",
    comment = "Auto-complete feature on papers search.",
    defaultValue = "true",
    constraint = "^(true|false)$",
    support = Config.FILE // Or: Config.ARGS or Config.SYS
  )
  private boolean autoComplete;

  public boolean isAutoComplete() {
    return autoComplete;
  }
}
```

Fig. 3 Illustration of requirement R1 (Options-as-Code) using the syntax of *Config2Code*, and the encapsulation of that option using the accessor *isAutoComplete* to respect requirement R2 (Encapsulation of Configuration Access).

```
for (int i = 0; i < EntryMarker.MARK_COLOR_LEVELS; i++) {
   Color c = Globals.prefs.getColor("markedEntryBackground" + i);
}
```

Fig. 4 Example of a dynamic configuration access.

In contrast, traditional configuration frameworks like the *Preferences* framework used in Figure 1 do not respect this requirement, since they treat configuration option names as string constant attributes, variables, or even string literals. On the one hand, this allows to dynamically construct option names through string concatenation. For example, the consumer class in Figure 4 accesses the options *"markedEntryBackground0"* to *"markedEntryBackground5"* by concatenating *"markedEntryBackground"* and the variable *"i"*. On the other hand, finding such a pattern of configuration options in the source code is not straightforward using regular textual search tools, basically complicating the traceability between option names and code, as well as other regular coding and maintenance activities.

Figure 3 illustrates requirement R1 on a Java application, using the syntax of our *Config2Code* prototype, for the *"autoComplete"* configuration option presented in Figure 1. The type of the option is determined by the type of the attribute (boolean), while the annotation *@Config* specifies a namespace for the option, a default value, constraint and the desired type of storage medium (*support*).

Apart from reducing the need for context switches and improving traceability, this first requirement also brings other benefits to configuration frameworks. For example, ownership of an option is now determined through code ownership of the source code file it is defined in. Furthermore, determining the impact of an option is now possible by reusing code analyses and tools originally designed to determine the impact of a variable. Those analyses and tools now allow to determine the (configuration) variables impacting the code location of a configuration failure [40] or to safely remove a configuration option

from the code without causing undesirable code paths to become active [2]. Finally, given that the definition of configuration options is now part of the code, any changes to such a definition will be captured in the version control system as regular code commits. Since these are the commits considered by code review, run-time configuration changes become an integral part of the review process instead of a special case [1].

R2. Encapsulation of Configuration Access

The second requirement focuses on the manner in which the values of options defined according to requirement R1 can be accessed by the rest of the code, in particular the ConsumerClasses in Figure 1. Based on common software engineering sense [12], this requirement states that **a configuration framework should provide a well-encapsulated API for accessing configuration values**, reducing coupling and duplication within the application. As shown in Figure 3, one needs to encapsulate the configuration option by making it private, and access it only via public accessors.

Requirement R1 makes the definition of a configuration option explicit in the source code, which allows the encapsulation of a configuration access (Requirement R2), as shown in Figure 2. For example, the *autoComplete* configuration option is explicitly defined in the source code via the annotated attribute *autoComplete* in the ProviderClass *JabRefPreferences*, which allows the encapsulation of that attribute using the accessor *isAutoComplete*. Note that it does not suffice to just adopt a third-party configuration framework like *jConfig* or *Preferences*, since scattered usage of such a framework's API throughout an application's code base leads to strong coupling and complicates later migration to another framework, as shown in the way the 5 "markedEntryBackground" options are accessed in Figure 4. In fact, these traditional frameworks do not enforce such encapsulation of options since options might be accessed by passing option names as literal strings throughout the whole source code.

Apart from reducing coupling between configuration and the application, encapsulated access results in uniform API usage throughout the application, again making it easier to determine the impact of a given option and whether an option can be safely removed. While requirement R1 has made the definition of configuration options explicit in code changes, the usage of these options across the code base now also is made explicit through a cohesive API, further encouraging systematic code review of configuration option-related changes.

R3. Generation of Configuration Media

While requirement R1 brought the definition of configuration options into the code and requirement R2 distributes the value of these options in a disciplined way, requirement R3 closes the loop by stating that **a configuration**

```
<configChecker>
  <module name="name">
   <property name="mandatory" value="1" />
   <property name="format" value="^[A-Za-z]{3,12}$" />
   <property name="type" value="error" />
   <property name="message" value="Incorrect name format" />
  </module>
</configChecker>
```

Fig. 5 Illustration of requirement R4 (Automatic Configuration Validation), showing checkstyle rules encoding programming conventions for configuration options, for the example in Figure 3.

framework should automatically generate the necessary configuration storage medium for the end user based on the source code. Requirement R1 eases the analysis of the source code to identify configuration options, which allows the automatic generation of a configuration storage medium (as shown in Figure 2). This medium, for example an XML file in Figure 1, a Java properties file or a .json file, or even a relational database, presents the current set of configuration options and their metadata (including a default value) to the end user. It is the location where the end user can change an option's value and it also is the data source from which the application reads the (possibly updated) option values.

Requirement R3 basically ensures that the configuration storage medium always is synchronized with the options that are currently used (defined) in the code by the developers, both in terms of option name, type, documentation and constraints. This automatic generation also allows end users (or DevOps engineers) to easily compare the previous version of the medium (containing the user's custom option values) to the new version in order to detect new options, removed options, changed constraints or option types, etc. In contrast, traditional configuration frameworks such as *Preferences* do not implement such a requirement and developers need to manually synchronize the source code and configuration option storage medium, which is risky and error-prone, as discussed in Section 3.

This requirement also allows the type of storage medium to be easily changed, and different types of media could be mixed for different subsets of the options. For example, one could use a configuration file for some options, while using command line arguments or a database for others. Furthermore, the requirement also achieves complete and automated traceability between options and code, and helps to address the challenge of missing option documentation, especially when combined with requirement R4.

R4. Automatic Configuration Validation

Requirement R4 states that **a configuration framework should automatically validate the definition of options as well as the values assigned to them**:

- Managers or technical leads can specify directives for configuration options that should be respected by developers, for example a specific naming convention. That can improve, as an example, the readability and comprehensibility of the generated configuration medium (R3), as shown in Figure 2.
- Each option should only accept values of a certain type, for example an IP address (4 numbers from 0 to 255) vs. a hostname (textual string).
- In addition to option types, an option should respect a range of values or constraints. For example, a memory size limit might not be higher than 1,024 MB.

Note that such validations are not feasible without an explicit definition of configuration options within the source code (Requirement R1: Options-as-Code), as shown in Figure 2.

This validation should be performed automatically, either during a build of a new version of the source code (developers) or during program start-up (end users). If a violation of a constraint is detected, the system should either halt or fall back to option values that are known to be good. Automatic constraint validation improves comprehension, and enforces naming conventions and the presence of documentation. In contrast, traditional frameworks such as *Preferences* do not respect Requirement R4, forcing developers to perform such validations manually or by writing additional scripts.

As an example, Figure 5 shows a static checkstyle rule for option names specified by the team lead of the example in Figure 3. The rule encodes that the name of an option is mandatory and should consist of 3 to 12 letters (upper- or lowercase), and it specifies an error message (not just a warning) in case of a violation. Furthermore, during compilation the type of the option will be enforced (since it corresponds to the type of a class attribute). At run-time, when the value assigned to an option is read from the storage medium, the constraint specified in the annotation shown in Figure 3 will be checked.

5 Config2Code Prototype

This section discusses the main components of the final prototype obtained at the end of the interviews, since it has been used subsequently to empirically evaluate the impact on configuration engineering activities of the 4 identified requirements. This prototype, which we named *Config2Code*, is a configuration framework implemented as a Java builder plugin that can be integrated into Eclipse or even into the Maven build tool.

5.1 Syntax

As shown in Figure 3, a developer specifies the metadata of a configuration option via the annotation `@Config`. For each option, one needs to specify a name, a namespace the option name belongs to, a comment to describe the option's goal to users, a default value, a constraint that the option should

```
; Auto-complete feature on papers search.
; Constraint = ^(true|false)$
papers.search.autoComplete = true
```

Fig. 6 Automatically generated INI file.

```
<property>
  <entry>
    <papers>
      <search>
        <!-- Auto-complete feature on papers search. -->
        <!-- Constraint = ^(true|false)$ -->
        <autoComplete>true</autoComplete>
      </search>
    </papers>
  </entry>
</property>
```

Fig. 7 Automatically generated XML file.

respect, and where the option is stored (e.g., in a configuration file, command-line arguments, or system configuration option). This meta-data respectively is defined via the `@Config` attributes `name`, `namespace`, `comment`, `defaultValue`, `constraint` and `support`.

To ensure a good encapsulation of configuration option access inside the code (requirement R2), one needs to define the option's attribute as private and provide a public getter.

5.2 Configuration Generator

This component automatically generates configuration storage media (typically textual configuration files) from the `@Config` annotations. Basically, this component reads all the information in the software annotations, then transforms those to a configuration file in a straightforward manner. For example, if an INI configuration file format is requested, *Config2Code* generates the .ini file shown in Figure 6. If, instead, a developer opts for an XML format, the configuration file shown in Figure 7 will be generated. Other types of configuration files like JSON can be easily supported.

5.3 Builder

The generation of configuration files is automatically performed during the build of the source code. In fact, "Config2Code" can be plugged into Eclipse and Maven software builders. *Config2Code* has a number of configuration options by itself, which basically allow to enable or disable features like the generation of configuration files and the use of injection (next section), and to override the configuration file format used.

5.4 Injection

To assign values to the class attributes that represent configuration options, *Config2Code* uses the principle of injection, based on aspect oriented programming principles. During the build of a software application, we modify its Java bytecode to call a synthetically generated injection method before executing the constructor of Provider classes. This injection method takes as arguments the class and attribute to instrument and the configuration option value to inject (as extracted from the generated configuration storage medium). As such, all option attributes will contain the right value by the first time their getter is called.

To modify the bytecode, we used the popular Javassist framework [41]. Since it is more lightweight than sophisticated injection frameworks like AspectJ or Guice, it makes integration into the build system of a software application more straightforward. In addition, AspectJ builders ignore annotation processors, and hence cannot be used for *Config2Code*.

5.5 Constraint Checker

This component checks the correctness of option values specified by users in the configuration storage medium by validating the constraints of the option when the application starts up. In the current version of *Config2Code*, each constraint is a single regular expression that developers define and is textually checked on the option values. While this works to some extent even for numeric data, dedicated constraint checkers and languages like Z3 would improve usability.

5.6 Checkstyle

In order to check whether the configuration options themselves (i.e., not their values) follow the naming and other conventions established by a project, *Config2Code* uses checkstyle rules specified in an XML file as shown in Figure 5. The resulting warnings and errors can then be displayed visually by the builder component inside the Eclipse IDE, as shown in Figures 8.

6 Design of User Study

Apart from identifying the four major requirements for configuration frameworks, this paper aims to empirically evaluate the degree to which they support developers in performing typical configuration engineering tasks. To this end, we carried out a controlled experiment [42], following the 5-step methodology of Meng et al. [43]. The design of this study is inspired by that of Wettel et al. [44].

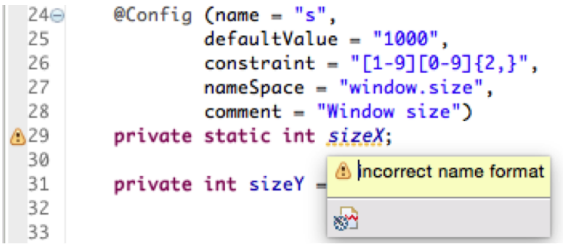

Fig. 8 Example of a checkstyle warning. The name of the configuration option *"s"* does not respect the naming convention is specified in the checkstyle file. A developer can configure checkstyle to either give a simple warning or abort the build upon violation of a rule, as shown in Figure 3.

6.1 Research Questions

Our controlled experiment compares a configuration engineering framework implementing the 4 requirements to a baseline state-of-the-practice framework in order to address the following two research questions across a wide range of typical configuration engineering tasks:

- RQ1: Do the requirements increase the correctness of configuration engineering tasks?
- RQ2: Do the requirements reduce the time needed to complete configuration tasks?

We analyze the results of these questions both quantitatively and qualitatively, and also consider the impact of participant experience (confounding factor). The remainder of this section presents the study object, then discusses the design of the experimental tasks, followed by the choice and composition of subject groups, and finally our experimental protocol.

6.2 Study Object

As study object, we looked for an open-source GUI application with a non-trivial number of configuration options that is not implementing the 4 requirements for configuration engineering. We focused on a GUI application such that configuration changes would easily be visible to study participants. Furthermore, the source code of the application should be large enough to be challenging, but should be structured well enough so as not to overly divert the participants' attention from the configuration.

Eventually, we selected JabRef [14], which is an open source tool dedicated to managing BibTeX references. Two of the authors are familiar with this software project from a previous study. JabRef 4.0 consists of 171,233 lines of code, 1,421 classes and 177 options. Hence, considering the size of the application, the number of configuration options is substantial. By default, JabRef uses the *Preferences* configuration framework, which is a popular [17], basic framework that comes bundled with the Java SDK since Java 4. It reads configuration op-

tions from a configuration file, and allows developers to access different types of configuration options (`String`, `int`, `double`, etc.) via an API. These methods take as argument a configuration name and a default value that is returned in case the accessed option is not declared in the configuration file.

The *Preferences* framework as used by JabRef was not implementing the 4 requirements:

1. Options are defined in textual configuration files, not inside the source code.
2. JabRef is strongly coupled to *Preferences*, as it calls the Preferences APIs (`getInt`, `getString`, ...) throughout the code with the name of the requested option as String argument (sometimes as literal, sometimes within a variable).
3. The textual configuration files are maintained and updated manually as the code evolves.
4. No validation is performed of options nor of their values.

In order to use JabRef for our study, we made two important changes. First, we removed JabRef's custom configuration GUI from the code base in order to allow subjects to focus only on the source code and the external (*Preferences*) configuration file. That JabRef GUI is simply an additional GUI that helps users to configure JabRef, but it is not related to a specific configuration framework. In other words, the custom configuration GUI is simply reading option values from the configuration file and writing back the user's changes to the same configuration file. Keeping the GUI would require participants to synchronize its source code with the changes of each configuration modification task, which would only add unnecessary complexity and possibly exceed a reasonable duration of the experiment. Second, we prepared a *Config2Code* version of JabRef by removing entirely the use of *Preferences* and replacing it with *Config2Code* annotations. As a result, all 177 *Preference* options were replaced by *Config2Code* options across the Consumer and Provider classes.

6.3 Task Design

The design of our experimental *Tasks* was driven by the 9 major activities of the run-time configuration engineering process [1], and their 13 challenges that we conceptually had mapped to the 4 requirements in subsection 4.1. While our initial goal was to formulate one study task per challenge, some challenges are related and we also wanted to keep the time for finishing all tasks below a reasonable duration of about 1h30. This is why we ended up with 7 tasks spread across 7 configuration engineering activities, basically skipping 2 activities. Since we split 3 tasks into two sub-tasks, we ended up with a total number of 10 tasks related to 7 configuration engineering activities.

The first skipped activity consists of managing the configuration storage medium, such as migrating from one configuration storage medium like XML to another one like a relational database. We skipped this task as migrating the usage of all options to a new storage medium is rather time-consuming to

Task	Description
T1. Creation of configuration options	Participants should create two new configuration options (one boolean **(T1.1)** and one integer **(T1.2)** option) based on requirements about the name of these options, their comments, default values, constraints for user choices, and the code area that should be modified to use the new created options.
T2. Refactoring - Changing a default value	Participants should change the default value of a configuration option. While this requires *Config2Code* users to change only an attribute of the *@Config* annotation, *Preferences* users need to change the code in two different places for this (the configuration file and a map data structure with default values).
T3. Refactoring - Removing configuration options	Participants need to remove 5 similar options and replace them by one unique option. *Config2Code* participants need to remove the "*@Config*" annotation, and change the corresponding class attributes' accessors to instead use the proposed option. *Preferences* participants need to remove the option from the configuration file, then search the code to find where each option is used to replace it by the proposed option. This task is not trivial, as some of the option accesses use Java reflection.
T4. Comprehension of options	Participants had to identify the range of possible values of two configuration options. The first option **(T4.1)** is a JabRef specific configuration option related to the auto-completion of textual fields in the GUI, whose allowed values are specified inside the JabRef code base. The second option **(T4.2)** is related to the possible font styles that can be used. The latter option's values can be found online, yet not all of those are actually used by JabRef. Participants were asked to only report the used ones.
T5. Fixing a configuration error	Participants need to identify which option is responsible for a configuration error (incorrect value), then to fix the error by preventing future errors of this option. This fix requires adding a "*@Config*" constraint for *Config2Code* or adding an if-check for *Preferences*.
T6. Configuration quality	The goal of this task is to check which options do not respect a predefined naming convention **(T6.1)** and to propose 5 examples of these options **(T6.2)**
T7. Configuration review	This task requires participants to review a patch of a newly created option, whose definition contains two problems: the option did not have any constraint attached (despite the commit message clearly mentioning the constraint), and the default value did not respect the constraint of the commit message.

Table 4 The 10 tasks administered in the user study.

consider in a single user-study. We also skipped the activity of managing (i.e., changing) option types, because it involves the same framework requirements as the other two maintenance related tasks (T2 and T3). Table 4 provides descriptions of the 10 resulting tasks, while Table 3 maps them to the 13 considered challenges. The handouts we gave to participants are available online [45, 46].

6.4 Participants

We initially designed our experiment to consider four categories of subjects, across two dimensions: {Industry, Academic} and {Novice, Expert} (Table 5).

	Industrial		Academia		All		Total
	Novice	Expert	Novice	Expert	Novice	Expert	
C2C.	4	3	17	5	21	8	29
Pref.	3	3	16	4	19	7	26
Total	7	6	33	9	40	15	55

Table 5 Decomposition of the user study subjects.

Novice industrial participants are subjects working in industry with less than 3 year of experience, while industry experts have more than 3 years of experience. Novice academic participants are undergrad engineering students, whereas expert academic participants are students pursuing a master or Ph.D.

However, one of the main barriers for controlled experiments of software engineering tools is participant recruitment [47], especially due to software professionals being busy [48]. To complement the 2 industry participants that we contacted based on personal contacts, we also contacted 5 remote freelance developers on Freelancer.com. We controlled for several well-known issues involving remote participants [48]. First, since developers tend to inflate their level of experience, we interviewed all candidates via chat and asked them to perform a warm-up task to measure and evaluate all skills requested from the developers. Only participants who correctly completed the warm-up participated in the experiment. Second, to deal with developers who may temporarily suspend a task or would not follow the instructions correctly, we required all online participants to record a video screencast during their working session. Third, high payments could attract participants that are excessively motivated by money, which may lead to unrealistic behaviour [48]. To mitigate this, we paid Freelancer.com's median flat sum of CAD$35 per experiment (in line with Ko et al.'s US$30 [48]) after the work was completed and we were 100% satisfied by its quality.

All of the academic participants were volunteers, which we invited to participate in exchange for a bonus in their courses and a certificate acknowledging their participation. The students were recruited at one North American and one North African university during the Summer of 2017 and Winter of 2018. None of them had prior experience with *Preferences* or *Config2Code*. In addition, we also had 7 student and one expert participants who participated in pilot runs of the study in order to refine the questions and study protocol. They are not included in Table 5, nor in the results section.

While our study design targeted the impact of both the {Industry,Academic} and {Novice,Expert} factors, we eventually dropped the former. First of all, we had an unbalance between industry and academic participants, with a ratio of 13 to 42. While this problem was not unsurmountable, we noticed that many students, even novices, had prior experience developing software. In some cases, students had more experience than (novice) freelancers. In one case, an academic expert (PhD student) had only 1.5 years of Java experience compared to an academic novice with 5 years of Java experience, while 3 industrial experts have 4 years of experience which is still less than the academic novice

with 5 years of Java experience. Therefore, to not derive wrong conclusions, we elided the {Industry, Academic} factor and only consider {Novice, Expert} in our discussion. This is why we eventually only consider the number of years of Java experience as a metric for experience (i.e., not grad vs. undergrad), i.e., a participant with 3 or more years of Java experience is considered as an expert, whereas participants with less than 3 years are novices. This decomposition is shown in the "All" column of Table 5.

6.5 Experimental Protocol

Participants were randomly assigned to either the experimental group (using *Config2Code*) or the control group (using *Preferences*). We used stratified sampling based on the {Expert, Novice} factor, resulting in the composition of Table 5.

The experiment was performed using two types of virtual machines (VMs): *(1)* VMs hosted on Google Cloud for non-local participants (mainly for the freelancers), and *(2)* a similar environment on VirtualBox installed on our lab machines for local subjects. Each VM was set up with Eclipse and either *Config2Code* or *Preferences*. To analyze the subjects' results, the screens of the VMs were recorded via the Cattura Google plugin or by configuring VirtualBox to capture the VM screen.

Before performing the 10 tasks, both the experimental and control groups received an introduction about configuration options in general, the specific framework they were going to use, and the three steps of the experiment (warm-up, experiment, and exit survey). We did not divulge our intent to compare *Preferences* and *Config2Code*. The participants then started with a warm-up exercise on a toy project that taught them how to use the framework they were assigned to. The results of this exercise were just used to filter out participants that were unfit for the study (not included in the numbers of Table 5).

Once finished with the warm-up exercise, the participants would enable screen recording of the VM and start the 10 tasks. We warned them that it might typically take 1h30 to finish all tasks, yet they were free to stop at any time. All sessions of experiments were supervised by one of the authors to enable clarification questions, if needed. 6 of the 10 tasks required a written answer in a separate response file saved in the desktop of each virtual machine. Once finished (or when quitting), the screen recording would be stopped, and the participants had to fill out an exit survey to express their impression about the experiment in general, and about the advantages and challenges they faced during the experiment related to the framework they used.

Finally, in order to address the research questions, we marked the modified source code and the answers in the response files of the 10 tasks to determine correctness. We defined a check-list to mark each task. For example, for T1.1 and T1.2, participants should (1) define the new options, (2) comment them, (3) define their constraints, and (4) use them within the source code. If all 4

elements would be there, they received the full mark (100%), while they would lose 25% for every incorrect element, resulting in a mark of 0 if they would miss all 4 elements.

In order to determine the time needed to perform the individual tasks, we scrolled through each participant's video to record the moments on which they switched to the next task. A task starts when a subject finishes reading its requirements and finishes when she completed the code or answered the exercise on the response file. The challenges we encountered were the length of the videos (up to 4 hours) as well as the fact that some participants answered some questions in more than one shot. They would start a given exercise and come back to finish it later on or even at the end of the experiment. In those cases, we summed up all periods during which they (visibly) were working on a question. Note that the rating process was made by two of the four authors.

7 Quantitative Results

This section discusses the results of our empirical evaluation of the impact of the 4 configuration framework requirements on the ability of developers to perform typical configuration engineering tasks. The next section then analyzes those results qualitatively for each individual task.

RQ1: Do the requirements increase the correctness of configuration engineering tasks?
Motivation: The 4 requirements discussed in section 4 are conjectured to enable developers to perform configuration engineering tasks more correctly compared to not following them. Therefore, we define our null hypothesis as:

H0: There is no significant difference in task correctness between Config2Code and Preferences participants.

In the qualitative analysis, we link this global hypothesis back to the individual requirements.

Approach: Based on the textual answers to each task and the modified source code of each participant [49], we assigned, for each task, a mark to the participant. Since subjects were free to leave at any point or skip any question, we ignore the tasks that they did not perform (instead of giving a zero score for those). For this reason, we analyze the marks of each individual task rather than calculating a global score, then we count the number of tasks for which we saw an improvement, deterioration or no visible difference when using *Config2Code* as opposed to *Preferences*. Similar to Wettel et al. [44], statistically significant differences in the number of improvements, deteriorations and cases without change were determined using the (non-parametric) Mann-Whitney-Wilcoxon test, with a confidence level of $\alpha = 0.05$.

In a second step, we evaluated the impact of participants' experience on the results (i.e., experts vs novices). For this, we compared the results of *Config2Code* experts against *Config2Code* novices on the one hand, and *Preferences* experts against *Preferences* novices on the other hand. This compar-

ison again is at task-level and based on Mann-Whitney-Wilcoxon tests with $\alpha = 0.05$.

Results: *Config2Code* outperforms *Preferences* in 7 (T1.1, T1.2, T2, T3, T4.1, T4.2, and T6.1) out of 10 tasks. Table 6 indeed shows how we can reject the null hypothesis ($p < 0.05$) for 7 out of the 10 tasks, while we were unable to do so for the other tasks. As shown by the median scores, there was no task for which the *Preferences* subjects performed more correctly than *Config2Code* subjects. *Config2Code* increased median correctness with up to 300% compared to the corresponding *Preferences* scores for task T4.2 (comprehension of an option related to font styles), in which *Preferences* participants ended up searching possible option values online, instead of just using the artifacts they had in hand (source code and configuration file). At the other extreme, for T7 (configuration review), more than half of *Preferences* were unable to perform the task correctly, leading to a median score of 0, compared to 1 for *Config2Code*.

As highlighted in Table 6, we found that, overall, the results are not impacted by the experience level of participants ($p \geq 0.05$), except in the case of T1.2 (creation of an integer option) and T4.2 (comprehension of an option related to font styles) for *Preferences*. The difference between T1.2 (creation of an integer option) compared to T1.1 (creation of a boolean option) is the need to check the correctness of the new option via a regular expression constraint. This was not easy for novice developers, since they lost time trying to identify where they should check the constraint within the relatively large code base, and how to implement a regular expression constraint in Java. Many *Preferences* novice participants had to check online how to use regular expressions. Similarly, T4.2 (comprehension of an option related to font styles) was not easy for novice developers, who typically did not have any concrete strategy to find all possible values of a configuration option, except for searching online. *Config2Code* on the other hand seemed to level out differences between novices and experts for those tasks.

RQ2: Do the requirements reduce the time needed to complete configuration tasks?

Motivation: The goal of this research question is to evaluate if configuration frameworks implementing the 4 requirements help developers perform configuration tasks faster. Therefore, we define our null hypothesis as:

H0: There is no significant difference in time required between Config2Code and Preferences participants.

Approach: For each task, we measured the time required by each participant, filtering out participants who did not successfully complete a given task. We defined "successful" completion as obtaining at least half of the marks for that task, i.e., a score ≥ 0.5. Note that this also led to filtering out participants who forgot to start or prematurely ended screencast recording, since no timing

		T1.1. Creation of a boolean option	T1.2. Creation of an integer option	T2. Refactoring - Changing a default value	T3. Refactoring - Removing configuration options	T4.1. Comprehension of JabRef specific option	T4.2. Comprehension of an option related to font styles	T5. Fixing a configuration error	T6.1. Number of options that do not respect a naming convention	T6.2. Examples of options that do not respect a naming convention	T7. Configuration review
Correctness	p-value impr. vs. deter. vs. no diff.	**1.96e-05**	**8.53e-06**	**9.84e-07**	**3.95e-07**	**8.25e-06**	**6.99e-08**	0.35	**0.042**	0.83	0.15
	Median *Config2Code* Score	1	1	1	1	1	1	1	0.75	1	1
	Median *Preferences* Score	0.625	0.75	0.5	0.5	0.5	0.25	1	0.5	1	0
	Median Improvement	+60 %	+33.33 %	+100 %	+100 %	+100 %	+300 %	0 %	+50 %	0 %	Inf %
	p-value Experience (*Config2Code*)	0.83	0.83	0.72	0.40	0.56	0.54	0.51	NaN	0.54	0.27
	p-value Experience (*Preferences*)	**0.035**	**0.00059**	0.18	0.29	0.10	**0.025**	1	0.56	0.35	0.64
	#*Config2Code* Participants	29	29	26	27	29	28	24	15	24	22
	#*Preferences* Participants	26	26	21	19	21	21	11	12	17	17
Time	p-value impr. vs. deter. vs. no diff.		**2.94e-05**	**0.015**	**0.020**	**0.0014**	**0.0025**	1	**0.0061**	0.23	0.44
	Median *Config2Code* Time (s)		1,559.5	40	408.5	37	24	312.5	171.5	57	564
	Median *Preferences* Time (s)		2,831	131	215	182	446	191	379	94	424
	Median Improvement		-44.91 %	-69.46 %	+90 %	-79.67 %	-94.62 %	+63.61 %	-54.75 %	-39.36 %	+33.02 %
	p-value Experience (*Config2Code*)		0.21	0.89	0.35	0.86	0.097	0.15	0.34	0.49	0.89
	p-value Experience (*Preferences*)		0.59	0.51	0.90	0.69	1	1	1	0.32	0.4
	#*Config2Code* Participants		24	23	20	22	22	14	12	16	9
	#*Preferences* Participants		17	15	9	8	4	7	5	11	5

Table 6 Summary of the RQ1 correctness and RQ2 time results, with significant p-values in bold (for $\alpha = 0.05$). NaN indicates that experts and novices of *Config2Code* obtained exactly the same results, while the median improvement values are relative to the *Preferences* results.

	%Constraint	%Comment	%Usage	%Default
Config2Code	3.44	3.44	27.58	0
Preferences	73.07	53.84	26.92	7.69

Table 7 Percentage of participants forgetting to add constraints, comments, usage sites and default values during the creation of configuration options (T1).

	T1	T2	T3	T4	T5	T6	T7
R1. Options-as-Code	+	+		+			
R2. Encapsulation of Configuration Access			+				
R3. Generation of Configuration Media	+	+	+	+			
R4. Automatic Configuration Validation	+					+	

Table 8 Impact of the 4 requirements on each task: T1 (Creation of configuration options), T2 (Refactoring - Changing a default value), T3 (Refactoring - Removing configuration options), T4 (Comprehension of options), T5 (Fixing a configuration error), T6 (Configuration quality), and T7 (Configuration review).

information was available for them. Finally, we also analyzed the impact of subjects' experience on the required time to solve a task.

We observed in the videos that most of the participants solved tasks T1.1 (creation of a boolean option) and T1.2 (creation of an integer option) in parallel. They typically added both configuration options either in the configuration file for *Preferences* or as two annotations for *Config2Code*, then changed the source code to use both configuration options. Therefore, we report a single time measurement for both.

Results: *Config2Code* outperforms *Preferences* on 6 tasks (T1.1, T1.2, T2, T4.1, T4.2 and T6.1), does not improve 3 tasks (T5, T6.2 and T7), while *Preferences* outperforms *Config2Code* in one task (T3). As shown in Table 6, *Config2Code* helps to save up to 94.62% of development time (T4.2. comprehension of an option related to font styles) compared to *Preferences*, while only for one task *Preferences* developers performed 90% faster than *Config2Code*. It is important to note that for this task (T3. refactoring - removing configuration options) the results for *Config2Code* in RQ1 were significantly better than for *Preferences* (median of 1 vs. 0.5). Most of the latter participants removed the options but did not update the reflective call to these options or did not remove the option from the configuration file.

For the other tasks, the results for RQ1 and RQ2 match each other. For example, *Config2Code* developers can immediately understand a configuration option, with a reduction of 79.67% and 94.62% of time (T4.1 and T4.2). *Config2Code* developers are also able to add an option in a median of 1,559.5 seconds (25.99 minutes) with *Config2Code* instead of 2,831 seconds (47.18 minutes) for *Preferences* developers. Finally, the RQ2 results are not impacted at all by the experience of developers.

8 Qualitative Results

This section qualitatively discusses RQ1 and RQ2 for each task. Table 8 summarizes the impact of each requirement on our user study tasks. We observe that all requirements have a positive impact on at least one task.

T1: Creation of Configuration Options: As shown in Table 7, 73.07% of *Preferences* participants did not add configuration constraints that check the correctness of the value of an option, compared to only 3.44% of *Config2Code* subjects. In addition, 53.84% of *Preferences* subjects forgot to comment their configuration options. Because JabRef defines a map data structure of default option values within its source code, the *Preferences* participants had to define the default values not only in the configuration file, but also in that defaults map. This is why 7.69% of *Preferences* participants forgot to define default values in both places.

Therefore, **requirement R1** (i.e., putting all option-related information inside the source code in one location) has had a positive impact on the correctness of adding a configuration option as well as on the time required (see Table 3). On the other hand, using a configuration option requires writing additional code in each ConsumerClass for both *Config2Code* as well as *Preferences* participants, which is why similar percentages of participants (30% and 33%, respectively) forgot usage sites of the new configuration options.

From the videos, we observed that both groups of developers liberally used copy-paste of existing option definitions. While *Config2Code* participants could just copy and modify annotations in one source code area (that they first had to find), the *Preferences* subjects had to perform many more steps across different code areas. We observed that they initially created the options in the configuration file, added the new option names as two global constants, then added the default values to the right map, in order to then use the options in the right files. Given the complexity of all these steps, we noticed substantial trial-and-error for this group. One *Preferences* participant confirmed: *P35: "Seems to have a lot of needless steps between getting a config value and using it"*, while one *Config2Code* participant found: *P33: "It's easy to add a config variable and assign it in the config.ini file"*.

The difference in percentage of developers adding constraints, together with the large number of erroneous options added by the *Preferences* group, show how **requirement R4** helps developers to ensure correctness of new options. One *Config2Code* participant confirmed in the exit survey that *P33: "It's also easy to give extra constraints on the different fields of config variables once you have a working example"*, against a *Preferences* participant who highlighted the *P9: "Need to add more validation on the configuration data as user[s] can give any value"*. Finally, **requirement R3** helped participants by automatically regenerating configuration files after each modification.

T2: Changing Default Value: Changing the default value with *Config2Code* requires only changing the attribute "defaultValue" within the ProviderClass for *Config2Code* subjects, while it requires changing the default map within the source code and the configuration file for the *Preferences* subjects.

Forgetting to change the default value in both places is error-prone, since JabRef seemingly would be using a different value than the one specified in the configuration file (i.e., the one listed in the map data structure).

Due to the context switches between the configuration file and the source code, 90% (19/21) of *Preferences* developers modified just the configuration file or just the defaults map, while this task was straightforward for 76% (20/26) of *Config2Code* participants, as they have to make changes only in a single place inside the code (**requirement R1**) and because *Config2Code* automatically updates the configuration file after a source code modification (**requirement R3**), which was indeed a feature that participants like and makes *P15: "The framework [...] quite practical as it automatically updates the configuration file whenever the source code is modified via annotations".*

T3: Removing Configuration Options: Participants had to replace 5 configuration options (O1...5) by another one (O6). An additional complexity (for both groups) was the fact that these options were accessed by reflection. The difference in correctness (and to some degree time) between both groups for this task was due to **requirement R2**. While *Preferences* accesses the configuration options from the ConsumerClass without any encapsulation, *Config2Code* uses the ProviderClass accessors, either directly or via a reflective call from a ConsumerClass.

Therefore, due to the lack of such encapsulation, *Preferences* users faced the problem of having to search the code for all usage sites (Consumer classes) of these 5 options, as well as to remove them from both the configuration file, the default value map, and their names from the global constants. In contrast, *Config2Code* developers just had to modify the ProviderClass accessors to use O6 internally, and physically remove the annotation `@Config` to avoid synchronizing the source code with the configuration file. One *Preferences* participant said: *P37: "some code refactoring tasks [require him] to analyze the code deeply".*

That said, we did observe in the videos that some *Config2Code* participants initially removed the "@Config" annotation, and the (previously configurable) class attributes and its accessors. However, this yielded exceptions due to the hidden reflective call. These *Config2Code* participants first tried to fix that unexpected bug, before realizing the simpler solution (modifying the ProviderClass accessors instead of removing). That explains the substantial slowdown compared to the *Preferences* group.

Similar to T2, *Preferences* users had to remove the 5 configuration options from the source code as well as the configuration file, while **requirement R3** helped *Config2Code* participants to avoid switching between different contexts (source code and configuration file), as they just have to remove `@Config` from the source code and **requirement R3** suggests to automatically update the configuration file.

T4: Comprehension of Configuration Options: As shown in Table 6, there is a significant difference in correctness between *Config2Code* subjects (who were able to find exactly the possible values within the *@Config* constraint, i.e., **requirement R1**), and the *Preferences* subjects, who had to

search where that option is used within the source code or even to use online documentation. This especially was clear for T4.2. While some participants found some values online, their responses were inaccurate, as many of these values are simply not used by JabRef. This would only be clear by exploring the source code, which is time-consuming. Due to the automatic synchronization of option information (**requirement R3**), *Config2Code* participants always found the up-to-date set of option values, not only from the attributes of `@Config` annotation but also in the configuration file.

T5: Fixing a Configuration Error: This task did not show any visible differences in either correctness or time. The videos showed how participants would usually start out by guessing a likely incorrect candidate option (in terms of its value) related to the bug symptoms that they faced. In this case, because the bug was related to the auto-complete functionality, most of the subjects started by searching configuration options that have the keyword "autocomplete" in their names, then manually inspected these options' corresponding values. **Requirement R1**, which is the only requirement addressing the challenges "debugging configuration failures is hard" and "lack of configuration debugging tools" in Table 3, clearly does not suffice in this context.

A special characteristic of the configuration error addressed in T5 is that it was a functional error without any exception or error message, only showing a graphical effect. In future work, we plan to investigate configuration errors with an explicit error message as well as performance-related configuration errors, both of which would allow source code inspection techniques to be used.

T6: Configuration Quality: This task showed a significant speed-up and correctness for *Config2Code* (T6.1), but did not show any difference for both correctness and time of T6.2. Our qualitative analysis of the videos showed that some *Preferences* subjects manually inspected each configuration option, whereas other participants copy and paste the whole configuration file in online regular expression checkers to solve this task. This is time-consuming compared to Config2Code, where developers could declare the regular expression within a checkstyle rule (cf. Figure 5; **requirement R4**). Indeed, one *Preferences* participant found that *P20:"The main problem is that there is no IDE validation of the keys used in the project (e.g., length, or naming conventions)"* in the *Preferences* framework. We also observed that only in one *Preferences* case, a developer wrote a script to identify the number of options that do not respect the proposed naming convention.

Due to the large number of configuration options that do not respect the configuration naming convention proposed in this task, it was not difficult (in terms of correctness and time) for both groups to identify a set of examples in T6.2, explaining the status quo for correctness and time.

T7: Configuration Review: Finally, we also did not observe any differences for the patch reviewing task. We believe that this is due to the limited size of the patch that we studied, making it an easier task for both groups. Furthermore, since the patch merely adds a new configuration option, all the information regarding the option's definition is included in the patch. A patch

that would impact code that depends on an option (a fact probably not obvious from the commit's diff alone) likely would not change other data of the option and hence might be harder to review. Future work should analyze this task in more detail.

9 Learned Lessons

This paper identifies and evaluates four principles to improve the development of configurable software systems.

Developers should consider options as code (Requirement R1). We have found that defining configuration options as code helps developers improve multiple configuration engineering activities, including the creation of configuration options, refactoring configuration options, and the comprehension of options. This is because this principle allows developers to define (for creation), change (for refactoring), or inspect (for comprehension) a single location in the source code to define all the meta-data related to a configuration option, without context (storage medium) switching. At the same time, this principle allows to exploit existing source code development tools on configuration options, for example to refactor or debug options.

The positive results for this principle confirm earlier results of, for example, Infrastructure-as-Code, which suggests that the principle could be extended even further to other software artifacts.

Developers should encapsulate the access to configuration options in one ProviderClass and use them via explicit accessors (Requirement R2). By encapsulating options in a single ProviderClass, developers have to change only that single place during their options refactoring. Our empirical study has shown how this avoids the refactoring challenges, particularly for removing or cleaning configuration options, typical [1] to systems with scattered access to configuration options. This principle is an extension of the traditional object oriented encapsulation practices, which might be extended to other software system artifacts.

Developers should automatically synchronize the configuration storage medium and the source code of a software system (Requirement R3). Requirement 3 helps developers to avoid inconsistent configuration option meta-data due to having to edit this data in both the source code and configuration option storage medium. We found that Requirement 3 has a positive impact on the creation and refactoring activities, since *Preferences* subjects missed to modify either the source code or configuration file. In addition, 53.84% of *Preferences* subjects missed commenting the newly added options during the creation of options (T1).

While requirements R1 and R2 correspond to developer practices, R3 (and to some extent R4) refers more to technical (automation) requirements. That said, the former two requirements make implementing the latter two requirements easier. For example, since developers do not need to maintain code and

configuration information in separate files (R1), generation of default configuration files from source code is more straightforward to perform.

Developers should automatically validate users' configuration choices (Requirement R4). Automatically ensuring the correctness of configuration options helps developers prevent failures due to unexpected values of a configuration option. While it is not that hard to develop option value validation functions, 73.04% of the *Preferences* subjects forgot to implement or call such a function.

Automatic configuration choice validation (Requirement R4) is related to the specification and enforcement of pre/post-conditions, but limited to the domain of configuration options. As suggested by R1, this requirement follows again from a similar guideline/convention in programming.

A development team should automatically enforce naming and other development conventions for configuration options (Requirement R4). Automatically verifying the quality of configuration options (e.g., their names and their descriptions) helps developers avoid substantial manual (and error-prone) effort. In our prior work [1], we found that 23% of surveyed developers have a naming convention, yet developers do not respect it. Interestingly, we observed that some *Preferences* developers wrote a script or used an online regular expression checker to find which options do not respect a naming convention, which is in other words an implementation of Requirement R4.

While we were able to address 10 out of the 13 challenges this paper focuses on, 3 challenges require additional configuration framework requirements. We observe that none of our four requirements addressed either the debugging of configuration failures or the reviewing of configuration options. Further studies are required to better understand how practitioners debug configuration failures and to propose additional configuration framework requirements to help them debug such failures. Such requirements might be based on existing configuration debugging approaches [6,50,51,52]. Similarly, we think that additional effort is required to understand how developers review configuration related patches and recommend configuration framework requirements for them.

10 Threats to Validity

Threats to *internal validity* concern alternative factors that could have influenced our findings. One threat is the degree of competence of our subjects. To mitigate this threat, we ensured that participants are comfortable with Java or at least had taken a Java course, in addition to recording their Java experience. Second, we used randomization to fairly assign participants to treatment groups. For each group, we presented the purpose of the study and provided a warm-up exercise to be sure they have a good knowledge of the application domain. Furthermore, participants from both controlled and experimental groups could ask questions during the experiments. Third, the subjects

may not have been correctly motivated. This threat was mitigated by the fact that all participants participated on a voluntary basis, receiving certain incentives (subsection 6.4). Fourth, some subjects might have been familiar with the popular Preferences framework compared to Config2Code subjects. To mitigate this risk, we provided a warm-up exercise to make sure that the participants get familiar with using both frameworks before performing their tasks. In addition, we evaluated the impact of participants' experience on our results.

Another threat to *internal validity* concerns the abandoning of some user participants or the absence of their screencasts, which respectively led to imbalanced data in RQ1 and RQ2. Although it would be better to have similar sample sizes between "Config2Code" and "Preferences", we were not able to control this as participants were allowed to abandon the experience at any stage they want, given the duration of the tasks that in one case took up to 3.53 hours. To mitigate this risk, we first used a stratified sampling approach and later used the Mann Whitney Wilcoxon test that is a good fit for imbalanced and small sample sizes.

Another *internal threat to validity* concerns the impact of confounding factors other than the 4 requirements on our findings. To mitigate this risk, we conducted a qualitative analysis of the video recordings and an exit survey with our subjects after finishing their tasks.

We also recognize a threat related to the design of the experimental tasks, which may have been biased to the advantage of either of the two groups [42]. To alleviate this threat, we aligned the design of tasks with the configuration engineering tasks and challenges identified in earlier work [1]. Two of the authors were part of a pilot study that assessed the perceived task difficulty and time pressure, then to validate that we are measuring the right metrics.

Threats to *external validity* concern the generalization of our findings. One major threat to *external validity* is the use of the generalizability of our results to other configuration frameworks. We used the framework *Preferences* as it is the default framework used by JabRef, it is one of the most popular frameworks [17], and especially because it does not implement our 4 requirements. However, the comparison against other frameworks might lead to different results. Hence, future work needs to compare the 4 requirements against other frameworks.

Future work should consider other activities and tasks than those studied here, covering configuration engineering activities and challenges not covered by the current study, and analyzing more in detail tasks related to the configuration reviewing activity. Furthermore, other object systems should be used, leveraging other standard configuration frameworks than *Preferences*. Finally, other participants should be recruited, involving more industry practitioners.

Threats to *construct validity* consider agreement between a theoretical concept and a specific measuring procedure. One threat considers the experimenter effect: since we are both the authors and experimenters, this may have influenced any subjective aspect of the experiment. Although we are aware that we cannot exclude all possible impacts of these threats, we did try to

mitigate them by designing a checklist and a model of the answers with a grading scheme. Moreover, two of the authors performed the grading.

Furthermore, the rapid prototyping process used to derive the core requirements, as well as the open coding process used to categorize the features within the final prototype, involve human judgement. To mitigate this risk, at least two of the authors were involved in both processes, furthermore we also observed that the new requirements to be added in the final prototypes (evaluated in the final interviews) started to saturate.

Finally, future iterations of this study should provide an automated way for participants to have the video started and stopped when starting or ending a task. This would avoid having to filter out participants without timing information.

11 Conclusion

This paper derives 4 requirements for configuration engineering tool support and empirically evaluates them via a user study with 55 participants and 10 tasks (spanning 7 configuration activities). Our findings show how the requirements can improve the correctness of 70% and the speed of 60% of the 10 tasks compared to a state-of-the-practice framework that does not implement the 4 requirements. We did not find any statistically significant differences for the other tasks, except for one in which *"Config2Code"* deteriorates in terms of speed but improves in terms of correctness. Furthermore, even novice developers benefited from the four requirements, as the improvements we observed in our study are independent of participant experience.

Through qualitative analysis of the study results (from the screencasts), we found that considering configuration options as code (**requirement R1**) by defining option meta-data inside the source code had a major positive impact on the participants' ability to perform configuration tasks swiftly and accurately. Coupled with automatic generation of configuration media (**requirement R3**) and configuration validation (**requirement R4**), and to some extent encapsulation of configuration option access within a clear Provider API (**requirement R2**), developers are able to deal with major configuration engineering challenges in their code base.

Finally, we suggest developers to consider their configuration as code as well as synchronizing the evolution of the source code with the configuration file. Future work should investigate on the evaluation of these principles on other artifacts that need to be synchronized with the evolution of the source code, such as the documentation of a software system, the build files, etc..

References

1. M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice - interviews, survey, and systematic literature review," *Transactions on Software Engineering (TSE)*, 2018. [Online]. Available: mcis.polymtl.ca/publications/2018/tse_mohammed.pdf

2. M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams, "Feature toggles: A case study and survey," in *Proceedings of the 13th IEEE Working Conference on Mining Software Repositories (MSR)*, Austin, TX, May 2016, pp. 201–211.
3. Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third Symposium on Operating Systems Principles*, ser. SOSP'11, 2011, pp. 159–172.
4. T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys*, vol. 47, no. 4, pp. 70:1–70:41, Jul. 2015.
5. T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'2015, NY, USA, 2015, pp. 307–319.
6. S. Zhang and M. D. Ernst, "Which configuration option should i change?" in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE'14, NY, USA, 2014, pp. 152–163.
7. M. Khan, Z. Huang, M. Li, G. A. Taylor, and M. Khan, "Optimizing hadoop parameter settings with gene expression programming guided pso," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 3, pp. 186–197, 2017.
8. D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *Proceedings of the 36th International Conference on Soft. Eng.*, ser. ICSE'14, 2014, pp. 215–224.
9. A. Sarma, G. Bortis, and A. van der Hoek, "Towards supporting awareness of indirect conflicts across software configuration management workspaces," in *Proceedings of the 22 International Conference on Automated Software Engineering*, ser. ASE'07, 2007, pp. 94–103.
10. F. Behrang, M. B. Cohen, and A. Orso, "Users beware: Preference inconsistencies ahead," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'15, New York, NY, USA, 2015, pp. 295–306.
11. S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proceedings of the 36th International Conference on Soft. Eng.*, ser. ICSE'14, 2014, pp. 140–151.
12. I. Sommerville, *Software Engineering*, 9th ed. USA: Addison-Wesley Publishing Company, 2010.
13. "Config2code," https://bitbucket.org/m-sayagh/config2code/.
14. "Jabref," http://www.jabref.org/.
15. R. Rabiser and D. Dhungana, "Integrated support for product configuration and requirements engineering in product derivation," ser. EUROMICRO'07, Piscataway, USA, 2007, pp. 193–200.
16. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.
17. M. Sayagh, Z. Dong, A. Andrzejak, and B. Adams, "Does the choice of configuration framework matter for developers? empirical study on 11 java configuration frameworks," in *Proceedings of the 17th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM'17, 2017, pp. 41–50.
18. L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, 1st ed. Addison-Wesley Professional, 2015.
19. K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 1st ed. The address: O'Reilly Media; 1 edition, 07 2016.
20. L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Relationship of devops to agile, lean and continuous deployment," in *Product-Focused Software Process Improvement*, P. Abrahamsson, A. Jedlitschka, A. Nguyen Duc, M. Felderer, S. Amasaki, and T. Mikkonen, Eds. Cham: Springer International Publishing, 2016, pp. 399–415.
21. S. Krum, W. Van.Hevelingen, B. Kero, J. Turnbull, and J. McCune, *Pro Puppet*, 2nd ed. Apress; 2nd ed. edition, 07 2013.
22. Ansible, "Core java preferences api," http://www.ansible.com/, Accessed March 08, 2018.

23. Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proceedings of the 12th Conference on Mining Software Repositories*, ser. MSR'15, 2015, pp. 45–55.
24. T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR'16, 2016, pp. 189–200.
25. P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: A systematic configuration validation framework for cloud services," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, NY, USA, 2015, pp. 1–16.
26. W. Li, S. Li, X. Liao, X. Xu, S. Zhou, and Z. Jia, "Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17, NY, USA, 2017, pp. 88–97.
27. Z. Dong, A. Andrzejak, D. Lo, and D. Costa, "Orplocator: Identifying read points of configuration options via static analysis," in *Proceedings of the 27th International Symposium on Software Reliability Engineering*, ser. ISSRE'16, 2016, pp. 185–195.
28. D. Jin, M. B. Cohen, X. Qu, and B. Robinson, "Preffinder: Getting the right preference in configurable software systems," in *Proceedings of the 29th International Conference on Automated Software Engineering*, ser. ASE '14, 2014, pp. 151–162.
29. S. Apel, D. Batory, C. Kästner, and G. Saake, "Feature-oriented software product lines: Concepts and implementation, berlin/heidelberg, 2013, 308 pages," ISBN 978-3-642-37520-0. URL http://www. springer. com/computer/swe/book/978-3-642-37520-0, Tech. Rep.
30. F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer Berlin Heidelberg, 2007. [Online]. Available: https://books.google.ca/books?id=PC4LyoSNNAkC
31. K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer Berlin Heidelberg, 2005. [Online]. Available: https://books.google.ca/books?id=lsX8_O_TRkEC
32. A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of configuration challenges in linux and ecos," in *Proc. of the 6th Int'l Workshop on Variability Modeling of Software-Intensive Systems*, 2012, pp. 149–155.
33. S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, Aug 2015.
34. F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 643–654. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884793
35. T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the 24 Symposium on Operating Systems Principles*, 2013, pp. 244–259.
36. A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," ser. ASE'11, Lawrence, KS, United states, 2011, pp. 193–202.
37. M. Sedlmair, M. Meyer, and T. Munzner, "Design study methodology: Reflections from the trenches and the stacks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2431–2440, Dec. 2012. [Online]. Available: https://doi.org/10.1109/TVCG.2012.213
38. S. Wagner and D. M. Fernández, "Analysing text in software projects," *CoRR*, vol. abs/1612.00164, 2015.
39. J. Cassoli, *Web Application with Spring Annotation-Driven Configuration: Rapidly develop lightweight Java web applications using Spring with annotations*, 1st ed. The address: CreateSpace Independent Publishing Platform, 10 2016.
40. X. Qu, M. Acharya, and B. Robinson, "Impact analysis of configuration changes for test case selection," ser. ISSRE'11, Hiroshima, Japan, 2011, pp. 140–149.
41. "Javassist," http://www.javassist.org/.
42. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction.* Norwell, MA, USA: Kluwer Academic Publishers, 2000.

43. X. Meng, P. S. Foong, S. Perrault, and S. Zhao, "5-step approach to designing controlled experiments," in *Proceedings of the International Working Conference on Advanced Visual Interfaces*, 2016, pp. 358–359.
44. R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 551–560.
45. "Config2code exercise," https://www.dropbox.com/s/8nfpnzhny46t9gc/ex_config2code.pdf?dl=0.
46. "Preferences exercise," https://www.dropbox.com/s/y88dneqqhsttkku/ex_preferences.pdf?dl=0.
47. R. P. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA'11, 2011, pp. 643–656.
48. A. J. Ko, T. LaToza, and M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, Feb 2015.
49. M. Sayagh and B. Adams, "Videos," http://mcis.polymtl.ca/ msayagh/userStudy/Config2Code/.
50. M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 255–265.
51. Z. Dong, A. Andrzejak, and K. Shao, "Practical and accurate pinpointing of configuration errors using static analysis," Bremen, Germany, 2015, pp. 171 – 180, automated debugging;Configuration errors;Configuration options;Correlation degree;Misconfigurations;Run-time information;Software program;Testing oracles;. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332463
52. S. Zhang and M. Ernst, "Automated diagnosis of software configuration errors," Piscataway, NJ, USA, 2013//, pp. 312 – 21, software configuration error diagnosis;ConfDiagnoser;configuration error identification;static analysis;dynamic profiling;statistical analysis;noncrashing configuration errors;configurable software system;Java;. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2013.6606577