Feature Toggles as Code: Heuristics and Metrics for Structuring Feature Toggles

Rezvan Mahdavi-Hezaveh[a,*], Nirav Ajmeri[b], Laurie Williams[a]

[a] *North Carolina State University, Raleigh, North Carolina*
[b] *University of Bristol, Bristol, United Kingdom*

Abstract

Context: Using feature toggles is a technique to turn a feature either on or off in program code by checking the value of a variable in a conditional statement. This technique is increasingly used by software practitioners to support continuous integration and continuous delivery (CI/CD). However, using feature toggles may increase code complexity, create dead code, and decrease the quality of a codebase.

Objective: *The goal of this research is to aid software practitioners in structuring feature toggles in the codebase by proposing and evaluating a set of heuristics and corresponding complexity, comprehensibility, and maintainability metrics based upon an empirical study of open source repositories.*

Method: We identified 80 GitHub repositories that use feature toggles in their development cycle. We conducted a qualitative analysis using 60 of the 80 repositories to identify heuristics and metrics. Then, we conducted a survey of practitioners of 80 repositories to obtain their feedback that the proposed heuristics can be used to guide the structure of feature toggles and to reduce technical debt. We also conducted a case study of the all 80 repositories to analyze relations between heuristics and metrics.

Results: From the qualitative analysis, we proposed 7 heuristics to guide structuring feature toggles and identified 12 metrics to support the principles

*Corresponding author
Email address: rmahdav@ncsu.edu (Rezvan Mahdavi-Hezaveh)

embodied in the heuristics. Our survey result shows that practitioners agree that managing feature toggles is difficult, and using identified heuristics can reduce technical debt. Based on our case study, we find a relationship between the adoption of heuristics and the values of metrics.

Conclusions: Our results support that practitioners should have self-descriptive feature toggles, use feature toggles sparingly, avoid duplicate code in using feature toggles, and ensure complete removal of a feature toggle.
Keywords: feature toggle, continuous integration, continuous development, open source repository, heuristic, metric

1. Introduction

Using feature toggles is a technique to either turn a feature on or off in program code by checking the value of a variable in a conditional statement. Feature toggles allow developers to integrate and test a new feature incrementally even if the feature is not ready to be deployed [1]. This technique is often used for continuous delivery (CD) and continuous integration (CI) in software development [1, 2]. Developers can also use feature toggles for other purposes, such as to perform experiments or to gradually roll out updates. Using feature toggles may impact the quality of the codebase. For instance, adding a toggle adds more decision points to the code, resulting in increased complexity. This increased complexity drives the need to remove feature toggles when their purpose is fulfilled. In 2012, developers in Knight Capital Group, an American global financial services firm, updated their algorithmic router which accidentally repurposed a feature toggle and activated dead code which had been unused for 8 years. Knight Capital Group lost nearly $400 million in 45 minutes, causing the group to go bankrupt [3]. This example shows the importance of structuring feature toggles correctly.

Developers may follow certain structures to incorporate feature toggles in their code. As an example, checking the value of a feature toggle could be done through different structures. One structure is to check the value of *all* feature

toggles through one method. As an alternative, each feature toggle could have its own specific method to check the value of that toggle. So, even a simple task of checking the value of feature toggles could be structured in more than one way. Feature toggles structured incorrectly could result in *technical debt* [4, 5]. Thus, arises a need for guidelines on how to structure and manage toggles. *The goal of this research is to aid software practitioners in structuring feature toggles in the codebase by proposing and evaluating a set of heuristics and corresponding complexity, comprehensibility, and maintainability metrics based upon an empirical study of open source repositories.* Software practitioners prefer to learn through the experiences of other practitioners [6]. To address the goal, we systematically study feature toggle usage in open-source software repositories, and (1) develop heuristics, (FT-heuristics), to guide the structuring of feature toggles; and (2) propose metrics, (FT-metrics), to support the heuristics. Accordingly, we state the following research questions:

RQ_H (heuristics): What heuristics can be used to guide the structuring of feature toggles in a codebase?

RQ_M (metrics): What metrics can be used to measure the effect of incorporating proposed heuristics in the codebase?

RQ_S (survey and case study): To what extent do the practitioners incorporate the heuristics, and how the metrics are related to those heuristics?

We answer RQ_M and RQ_H iteratively and concurrently. To address the heuristics research question (RQ_H), we analyze the code base of 60 GitHub repositories and develop heuristics for incorporating feature toggles in the code. To address the metrics research question (RQ_M), we analyze the same 60 GitHub repositories to identify metrics that support heuristics. Specifically, we analyze files in these repositories, study existing design metrics, such as CK metrics [7] and software product line variability metrics [8], and select the metrics that can be related to feature toggles. To address the third research question (RQ_S), we conduct a survey of practitioners in the 80 repositories in our dataset. Through the survey, we identify how difficult is it for practitioners to manage feature toggles, and to what extent they agree that FT-heuristics can be used to guide

practitioners on how they can structure and use feature toggles so as to reduce technical debt. We also conduct a case study on the same 60 GitHub repositories and additional 20 GitHub repositories (all 80 repositories in our dataset) based on the adoption of FT-heuristics and FT-metrics to analyze the relation between FT-heuristics and FT-metrics statistically.

We summarize the contributions of this paper:

(1) Development of 7 FT-heuristics to guide the structuring of feature toggles;

(2) Identification of 12 FT-metrics to support the principles embodied in the FT-heuristics;

(3) A case study analyzing practitioner adherence to the FT-heuristics in open source software, and the relation of FT-heuristics with FT-metrics; and

(4) A database of GitHub repositories that use feature toggles in their development cycle, and examples of good and bad structuring of the feature toggles in these repositories.

Organization. Section 2 provides a background of feature toggles and describes related works. Section 3 explains our research method. Section 4 describes FT-heuristics with examples. Section 5 details FT-metrics. Section 6 presents the results of the case study and the practitioner survey. Section 7 discusses the limitations of our study. Section 8 concludes with lessons learned and future directions.

2. Background and Related Works

This section describes the background and relevant related work.

2.1. Background

Rapidly releasing valuable software leads companies to use CI and CD to make development cycles shorter. CI is a practice to integrate, and automatically build and test software changes in a source repository after each commit

[9] in short intervals. CD is a practice to keep software in a state such that it can be released to production at any time [10]. Using feature toggles is a technique used by companies practicing CI and CD [2, 11].

The language constructs to implement feature toggles have long been included in programming languages. However, the first documented use of feature toggles to support CI/CD was at Flickr in 2009 [12]. Listing 1 is an example of a feature toggle usage where the value of the toggle `useNewAlgorithm` is checked to determine which search algorithm to call. If the value of the toggle `useNewAlgorithm` is `True`, the search function calls the new search algorithm, otherwise calls the old search algorithm [13].

```
function Search() {
    var useNewAlgorithm = false;
    if(useNewAlgorithm) {
      return newSearchAlgorithm(); }
    else {
      return oldSearchAlgorithm(); } }
```

Listing 1: An example of a feature toggle [13].

Researchers and practitioners have classified feature toggles in five types [14, 15, 16, 1]: (1) *release toggle* to enable trunk-based development, (2) *experiment toggle* to evaluate new features, (3) *ops toggle* to control operational aspects, (4) *permission toggle* to provide the appropriate functionality to a user, such as special features to premium users, and (5) *development toggle* to turn on or off developmental features. In our analysis, we do not differentiate between these five types.

2.2. Related Work

In this section we explain related works on feature toggles, configuration options, and metrics.

2.2.1. Feature Toggles

Parnin et al. [2] published 10 best practices from a discussion with researchers and practitioners from 10 companies. One of these best practices—*dark launching*—is enabled by feature toggles to incrementally deploy code into production but keep the new code invisible from users.

Rahman et al. [1] performed a thematic analysis to understand the challenges, benefits, and cost of using feature toggles in practice. They quantitatively analyzed feature toggle usage across 39 releases of Google Chrome over 5 years. Rahman et al. reported three objectives of using feature toggles: rapid release, trunk-based development, and A/B testing. They focused on the definition and usage of feature toggles over time, but we focus on structuring and incorporating feature toggles in the last snapshot of the repositories. Whereas their study was limited to analysis of Chrome's version history, our study focuses on qualitative analysis of structuring feature toggles in open source repositories on a larger scale.

In another study, Rahman et al. [17] extracted four architectural representations of Google Chrome: (1) conceptual architecture; (2) concrete architecture; (3) browser reference architecture; and (4) feature toggle architecture. Their study showed that using the extracted feature toggle architecture, developers can find out which module is affected by which feature, and vice versa. Whereas Rahman et al.'s study focused on the illustration of using feature toggles in understanding the modular architecture of the system, our study focuses on the structures of incorporating feature toggles in a code of the system.

Mahdavi-Hezaveh et al. [13] analyzed grey literature artifacts and academic papers on feature toggles and identified 17 feature toggle usage practices in 4 categories. Although practices such as "Use naming convention" and "Create a clean-up branch", identified by them, can reflect in the source code, they did not inspect the codebase which is our focus.

Ramanathan et al. [18] developed a refactoring tool to delete old and unused feature toggles in the code. Their tool analyzes the abstract syntax tree of the code, generates a diff on GitHub repository, and assigns it to the author of the feature toggle. Meinicke et al. [19] proposed a semi-automated approach to

detect feature toggles in open-source repositories, by analyzing the repositories' commit messages. They found 100 GitHub repositories that use feature toggles via keyword search in commits. Meinicke et al. analyzed some aspects of feature toggle usage in these identified repositories, such as the relationship of having short-lived toggles and having a toggle owner. Automation tools and approaches can be improved considering FT-heuristics and FT-metrics.

2.2.2. Configuration Options

Configuration options are key-value pairs to include or exclude functionality in a software system [20]. Despite feature toggles and configuration options being similar concepts, they have distinguishing characteristics, such as their users and lifetime. Whereas configuration options are used by end-users and can exist permanently, feature toggles are used by developers and are ideally removed from code. However, in reality, a large fraction of feature toggles stay in the code base forever [19]. Based on the definition of configuration options and feature toggles by researchers in the literature [20, 1, 14], configuration options can be considered as a subset of the feature toggles (including ops toggles and permission toggles) or feature toggles can be seen as configuration options that are used for new purposes (including release, experiments, and development toggles). The concepts of configuration options and feature toggles are not distinguished clearly in the research literature. In this study, based on [20], if the value of a feature toggle could be changed by end users, we consider it as a configuration option.

Sayagh et al. [21] interviewed 14 software engineering experts, surveyed Java software engineers, and conducted a literature review on configuration options to understand practitioners' process of using configuration options, the challenges they face, and the best practices that they could follow. One of the identified activities in the process of using configuration options is *Quality Assurance* which refers to improving software configuration *comprehensibility*, reducing software configuration's *complexity*, and improving its *maintainability*. We consider this

categorization in FT-metrics.

Meinicke et al. [22] analyzed highly-configurable programs' traces to identify interactions among configuration options. These interactions impact the quality assurance of the programs as configuration space can grow exponentially. If structured incorrectly, this concern applies to feature toggles too. Whereas Meinicke et al.'s focus was interactions in configuration options, our focus is on structuring feature toggles.

Zhang et al. [23] studied 1,178 configuration-related commits of four open-source cloud system repositories. They analyzed the evolution of configuration design and implementation in these systems. Zhang et al.'s goal was to understand developers' practices to revise the design and the implementation of configurations from code changes in response to misconfigurations. Whereas they studied commits from four cloud system repositories over a period of 2.5 years and focused on revision of misconfigurations, our study analyzes 80 repositories from various domains and our focus is structuring feature toggles in the code.

2.2.3. Metrics

Chidamber et al. [7] developed and empirically validated a suite of six metrics (CK metrics) for object-oriented design. These metrics can be used to measure the object-oriented software development process improvement. The main focus of developed metrics is to measure the complexity in the design of classes. The six metrics are: (1) Weighted Methods Per Class (WMC), (2) Depth of Inheritance Tree (DIT), (3) Number of Children (NOC), (4) Coupling between object classes (CBO), (5) Response For a Class (UFC), (6) Lack of Cohesion in Methods (LCOM). Although some of our metrics overlap with CK metrics, Chidamber et al. focused on object-oriented software development which is not the focus of this paper.

Liebig et al. [8] analyzed 40 open-source software projects that use C preprocessors (cpp) to implement variable software. Using cpp is a popular approach

to implement configuration options. Liebig et al. introduce several metrics to measure cpp usage in terms of comprehension and refactoring, such as Lines of Feature Code (LOF) and Granularity (GRAN). Based on their results, Liebig et al. suggested alternative implementation techniques. Although some of our metrics overlap with Liebig et al.'s metrics, Liebig et al. focused on projects written in C. We do not filter projects based on programming languages. Our findings are language-independent. As we will discuss in Section 3.2, we consider these metrics, and via an iterative process, select the ones that can measure the effect of following FT-heuristics in a repository.

Although some of our findings may be applicable to structuring configuration options, we focus on improving the structuring of feature toggles. We conduct a large-scale qualitative analysis of a set of 80 repositories. Our findings are programming language independent.

3. Methodology

Figure 1 summarizes our two-phase method to develop the FT-heuristics and FT-metrics and the resulting dataset. Phase 1 address RQ_H and RQ_M, and Phase 2 addresses RQ_S.

3.1. Dataset–Repositories

We analyzed GitHub repositories that incorporate feature toggles. First, similar to Meinicke et al. [19], we searched GitHub repositories for the keyword "feature toggle". Our search date was 23-May-2019. GitHub categorizes search results into different categories. We inspected the search results in the following categories: (1) Repositories, (2) Code, (3) Commits, and (4) Issues. After inspecting the first 10 results in each category, we found that the results in the "Commits" category were the most appropriate for our study. GitHub search skips forks by default, so the commits of the forked repositories are excluded automatically. Including forks could skew the results. Search results in "Repositories" and "Code" categories listed repositories of feature toggle management systems which are not the focus of this work. Results under "Issues" can also

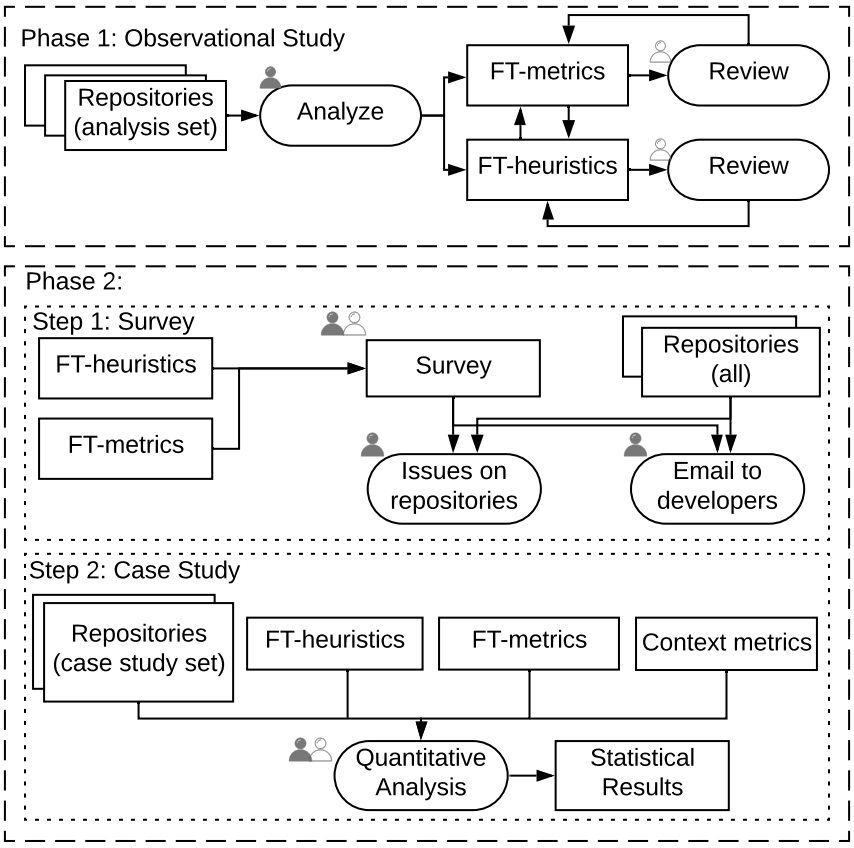


Figure 1: Research methodology outline.

be considered as a source to find repositories that use feature toggles. For this study, we used results in the "Commits" category but future work could consider using "Issues".

GitHub search returned approximately 465,000 commits with "feature" and "toggle" in their commit messages. By default, GitHub sorts the search results by *best match*, which we found to be appropriate for our search criteria. On inspection, we found the first 400 search results were most likely relevant to feature toggles. We manually examined each of the 400 commits including

commit messages, the files which were changed, and the changes which were made to the code; as well as the issues, pull requests, and documentation in each repository to find details about incorporating feature toggles.

We identified 110 relevant commits after excluding commits that met at least one exclusion criteria: (1) Commit URLs that no longer exist; (2) Commits related to toggle/button input controls in the user interface (UI) of an application. For example, the commit message is "add toggle-all feature" and the change is "added a checkbox to check all the options in UI". However, wrapping a UI element with a feature toggle is not excluded; (3) Commits in repositories of feature toggle management systems which are not in the scope of this study; and (4) Commits from the repositories in which we cannot distinguish feature toggles from configuration options. For example, if a toggle is defined in a way that end-users can change its value, we exclude the repository.

The selected 110 commits belonged to 80 repositories that use feature toggles in their development process. We list the language and lines of code (LOC) of these repositories in Table 1.[1]

Table 1: Characteristics of the identified repositories.

	Language	# repositories	LOC range
Top Five	TypeScript	17	7,840 to 68,583
	Java	15	227 to 361,213
	JavaScript	13	10,300 to 783,301
	PHP	8	1,567 to 400,034
	C#	7	18,232 to 148,390
Other languages		20	170 to 3,547,167
Total		80	170 to 3,547,167

[1] Data availability: The data including commits, repositories, and case study details is posted here: `https://sites.google.com/view/feature-toggles-dataset/`.

We randomly selected 60 repositories as the *analysis set* which was analyzed to develop the FT-heuristics and to identify the FT-metrics. *The case study set* consisting all 80 repositories (60 repositories from analysis set and 20 additional repositories) to find the relation between the FT-heuristics and the FT-metrics.

3.2. Phase 1: Observational Study

Figure 1 (top block) outlines the steps in Phase 1. To address RQ_H and RQ_M, we manually developed FT-heuristic and FT-metrics iteratively and concurrently through an observational study of the repositories in the analysis set. We looked for structural patterns and developed a mental model. The FT-heuristics help in the actionability of FT-metrics. The manual analysis for each repository contained the following steps:

(1) Starting with the GitHub commit we used to identify the repository, including the commit message, changed files, and changed lines of code in the commit, we identified the feature toggle configuration file and the feature toggle class. If the configuration file was not found in changed files, we searched the repository for the feature toggle that existed in the commit and traced it to find the feature toggle configuration file, which contains a list of toggles.

(2) We searched for the usage of all feature toggles identified in Step 1 in the code and commit history;

(3) We inspected issues, pull requests, documentation, and comments to find information about incorporating toggles.

(4) We recorded the details of structures of incorporating feature toggles observed in Steps 2 and 3, including definition details (e.g. file of definition, meta-data, names); usage details (e.g. checking value of toggle); removal details (e.g. removed lines of codes, changed files).

We develop the FT-heuristics to structure feature toggles using notes recorded in Step 4 of the above process. Following the open coding technique [24], we assign codes to observational notes from Step 4 and define an FT-heuristic for each category of codes based on our developed mental model. For example, one part of notes for each repository is about how they check the values of feature toggles with methods. We observe two codes in that note "Check all the values with one method" and "Check the values with specific methods for each toggle". We observed less complexity and less maintainability effort in repositories that use a shared method for checking feature toggle values. So, we define Heuristic 1 (SharedMethod) based on these codes and our observations.

The FT-heuristics embody the actionable recommendations based on the current state of using feature toggles in analyzed repositories. In addition, based on the notes, FT-heuristics, and considering relevant literature on metrics such as CK metrics [7] and metrics to measure variablity in software product lines using C preprocessors to implement configuration options [8], we identified FT-metrics to support FT-heuristics iteratively.

During this iterative process, we had the list of the metrics from the literature [7] and [8]. Although we did not have pre-defined criteria, we discussed the applicability of each one of the metrics to observed details of incorporating feature toggles in repositories iteratively by the first two authors. For example, metrics "Depth of Inheritance Tree (DIT)" and "Number of Children (NOC)" from CK metrics have no connection with feature toggles based on authors' observations. But, the concept behind metric "Lines of Feature Code (LOF)" from variability metrics is used in our "Feature toggle lines of code (M9)" metric. The discussions between the two authors help to reduce the subjective selection bias of the metrics. In the end, the metrics with no effect by incorporating feature toggles were removed from the list. Note that, not all metrics came from literature. We also added other metrics to the final list during this iterative process based on our observations, such as the presence of guidelines and the presence of duplicate code. Then, we categorized FT-metrics based on their effect and existing categories in literature [21] in three categories: *complexity*,

comprehensibility, and *maintainability* using card sorting technique [25]. Card sorting technique for categorizing FT-metrics is done by the first two authors of the paper. We pre-define these three categories and discuss the correct category for each FT-metrics. In this discussion, we rely on our observations from analysis repositories in this study and our experience as software engineers. When both authors agree on a category for a FT-metric, we move forward to the next metric.

The first author performed Phase 1 in consultation with the second author, who critically examined the definition and examples of the FT-heuristics, examined the definitions and measurements of the FT-metrics, and gave feedback in all steps. Sections 4 and 5 presents the results of Phase 1.

3.3. Phase 2: Survey and Case Study

The bottom block of Figure 1 outlines the two steps in Phase 2, which we follow to address RQ_S.

Step 1 (Survey): To evaluate the acceptance of our FT-heuristics by practitioners of all repositories in our dataset, we conducted a survey. In the survey, we asked to what extent do practitioners agree that the FT-heuristics can be used to guide practitioners on how to structure and use feature toggles to reduce technical debt. We used Likert scale [26] to specify the level of agreement: Strongly disagree to Strongly agree. [2] To distribute the survey, first we submitted *issues* in the repositories that allowed us to submit an issue. Second, we sent emails to practitioners when an email address of a feature toggles' contributor was available.

Step 2 (Case Study): To find the relation between FT-heuristics and FT-metrics, we analyzed the default branch of each randomly selected GitHub repository in the case study set.

First, using the commit by which we identified the repository, we (1) found the list of toggles in the repository; (2) measured each of the identified FT-met-

[2] Published data includes the survey questionnaire.

rics; and (3) checked whether the repository follows each of the FT-heuristics. Additionally, we gathered the following context metrics for each repository: (1) lines of code; (2) language; (3) number of contributors; and (4) number of feature toggles. We used the *cloc* [27] to calculate lines of code and identify the language. Using the GitHub profile of each repository, we identified the number of contributors for that repository. The number of toggles was identified via manual inspection of the repository.

Next, to examine the relationship between the FT-heuristics and the FT-metrics, for each heuristic, we separated repositories in two groups based on whether they follow (F) or do not follow (NF) that heuristic.

To compare the differences between the metric values yielded by the following (F) and not following (NF) groups of repositories, we define a measure named *Improvement* in Section 6.3. Improvement is computed as the percentage improvement in a FT-metric by following a FT-heuristic. Using the improvement measure, we discuss the trends in case study repositories.

In Appendix A, we report preliminary statistical analyses on case study data. This analysis includes regression analysis using the *Best Subset Selection* [28] (linear regression models for numeric metrics and logistic regression models for binary metrics) for each FT-metric.

The first author collected data, and the first and second authors performed statistical analysis of the results. We report the result of the survey; and the observed relation between FT-heuristics, FT-metrics, context metrics in Section 6.

4. FT-Heuristics

We now describe FT-heuristics which we derive by analyzing 60 "analysis set" repositories in Phase 1 of our method. For each FT-heuristic, we provide examples found in the repositories which follow or not-follow that heuristic. Our naming convention for the example is H (for Heuristic), followed by the heuristic number, followed by the subscript FE if the example is the following example,

and NFE if it is a not-following example. The number at the end of the subscript is a counter of examples for that heuristic. For instance, $H1_{FE1}$ means the first following example for Heuristic 1. The number in the parenthesis in front of each FT-heuristic name (subsection title) is the number of the repositories that follow the heuristic. Note that a low number for a heuristic does not necessarily correspond to low importance for that heuristic. A low number could be a sign of a related bad smell. For instance, we are aware that not having test cases is a bad smell but only 17 repositories follow H6 (Testing).

The derived FT-Heuristics are similar to general software engineering best practices. However, we find that these are not followed by developers in all repositories in the analysis set. So, increasing the awareness of the developers about the impact of following these heuristics is important.

4.1. Shared Method to Check Value (26)

Heuristic 1 (SharedMethod). Using one *shared method* to check the value of all feature toggles, instead of having a method to check the value for each feature toggle, can help reduce complexity and increase maintainability.

The value of a feature toggle is checked in a conditional statement of code. One approach to access the value of a toggle is to call a feature toggle value checking method from the feature toggle class, which is often named `isEnabled()` [13]. The lower the number of feature toggle value checking methods, the lower the code complexity. Having fewer feature toggle value checking methods also decreases (1) the number of files and the lines of code that need to be modified to implement and maintain a feature toggle; and (2) the probability of the presence of dead codes when deleting feature toggles because an associated feature toggle value checking method does not need to be deleted.

$H1_{FE1}$: In Listing 2, the name of the toggle is passed to the method and the value of the toggle is checked in the list of feature toggles [29]. When adding a feature toggle, the developer should add the toggle only to the configuration file or database. By doing so, the toggle can be used anywhere in the code. Adding the toggle to the configuration file minimizes the number of modified files and

lines of code. For removal, the toggles need to be deleted from the configuration file or the database and the part of the code where it is used. No modifications are needed to the value checking method.

```
export const isFeatureEnabled = (feature:Feature) =>
  (window as any).appSettings[feature] === 'on' || (window as any).
    appSettings[feature] === 'true';
```

Listing 2: One shared IsEnabled value checking method [29].

H1$_{\mathbf{NFE2}}$: The code in Listing 3 shows each feature toggle having its own function to check its value [30]. When developers want to define a new feature toggle, instead of adding a toggle and its value to the configuration file, they define a new customized `isEnabled()` function in the file contains all other `isEnabled()` functions. The number of files which should be changed is one, but the number of lines of code that should be changed is larger compared to H1$_{\mathrm{FE1}}$.

```
public bool IsAggregateOverCalculationsEnabled() {
        return true; }
```

Listing 3: IsEnabled function for one toggle [30].

4.2. Self-Descriptive Feature Toggles (19)

Heuristic 2 (SelfDescriptive). Using intention-revealing names for toggles, adding a description field in the configuration file as a meta-attribute for each feature toggle, and including comments when using the toggles can improve comprehensibility.

Having self-descriptive code is a known practice in software development [31, 32]. Self-descriptive code improves understanding and reduces code maintenance effort. Also, Sayagh et al. [33] suggest having self-descriptive configuration options. Feature toggles may remain in the codebase for a while and should be treated similarly to implementation code. For example, adding comments is a way to make code understandable.

H2$_{\mathbf{FE1}}$: As Listing 4 shows, each toggle in the configuration file of CFS-Frontend repository of the UK Education and Skills Funding Agency [34] has

an intention-reveling name and a description which makes the purpose of the toggle clear.

```
"EnableCheckJobStatusForChooseAndRefresh": {
  "type": "bool",
  "metadata": {
    "description": "Enable checking calc job status prior to
    choosing and refreshing" },
  "defaultValue": true }
```

Listing 4: A feature toggle with description [34].

H2$_{\mathbf{FE2}}$: A repository of Automattic [35] uses intention-revealing names and comments to explain code related to feature toggles. Two of these example comments (pertaining to `autorenewal` toggle in the code) are: *"The toggle is only available for the plan subscription for now, and will be gradually rolled out to domains and G suite"* and *"remove this once the proper state has been introduced."*

H2$_{\mathbf{FE3}}$: In the configuration file of a Salesforce repository [36], the feature toggles are grouped in two groups: *long term toggles* and *short term toggles*. The following is the description developers provided as a comment: *"Defining a toggle in either 'shortTermToggles' or 'longTermToggles' has no bearing on how the toggle behaves—it is purely a way for us to keep track of our intention for a particular feature toggle. It should help us keep things from getting out of hand and keeping tons of dead unused code around."* For adding short term toggles the comment is *"add a new toggle here if you expect it to just be a short-term thing, i.e. we'll use it to control the rollout of a new feature, but once we are satisfied with the new feature, we'll pull it out and clean up after ourselves."* In addition, this team uses intention-revealing names for feature toggles, and the configuration file is well commented.

In a survey study [13], practitioners suggested to "Determine the type of the toggle" before adding it to the code. When developers specify if the feature toggle is a short-lived toggle or a long-lived toggle, they can plan to remove the toggle at an appropriate time. Later, if developers need to limit the number

of feature toggles in the code, they have a list of short-lived toggles which can potentially be removed first from the code.

H2$_{\text{NFE4}}$: Developers of HMCTS [37] named a feature toggle `FEATURE_TOGGLE_520` which doesn't convey its purpose.[3]

4.3. Guidelines for Managing Feature Toggles (10)

Heuristic 3 (Guidelines). Providing *guidelines* for adding or removing feature toggles can improve comprehensibility and maintainability.

Management of feature toggles, including adding and removing toggles, is a challenge for developers and project managers. If feature toggles are added arbitrarily, a large number of toggles may end up in the code after a while. Developers should know when to add a feature toggle (For every new feature? For every huge change?) and when to delete a feature toggle (In a month? After publishing a new release?). Hence including guidelines for adding and removing toggles is important. Dead code may be introduced if the developers do not know when or how to remove a toggle correctly.

An example of feature toggle management is using pull requests to remove a toggle [13]. We observed that developers use issues and pull requests to add and remove toggles. Development teams may use other project management systems, such as a Kanban board, or wiki pages [21] to manage toggles, however, these are not tightly integrated with the code base and may be missed by developers.

H3$_{\text{FE1}}$: In a repository of The Guardian [38], developers use pull requests to delete a feature toggle. Developers can use *"feature toggle in:title"* as search string in the list of issues and pull requests of repositories to find those related to toggles.

H3$_{\text{NFE2}}$: In a repository from Australian Department of Veterans' Affairs [39], the guideline for adding feature toggles is provided in the `README` file.

[3]This feature toggle is now removed from the code. The link to the removing commit is `https://bit.ly/34tcjOk`

Although the developers have guidelines for adding toggles, they do not have guidelines for removing toggles.

4.4. Use Feature Toggles Sparingly (53)

Heuristic 4 (UseSparingly). Using a feature toggle in *as few locations as possible* in the code can reduce complexity and improve maintainability.

Having more locations to edit makes using feature toggles harder for developers. The additional number of files to update causes an increase in the development effort and the possibility of creating dead code. The more number of paths in the code, the higher the code complexity. Note that, the focus in this FT-heuristic is not to minimize the "number" of the feature toggles, but the count of files that a toggle is used in them is better to be as low as possible.

$H4_{FE1}$: Feature toggles could be either checked directly in conditional if-statements, or be used to set the value of a variable, and then the new variable could be checked or used in the rest of the code [1]. Listing 5 shows an example of using feature toggles to set the value of another variable. Instead of individually checking the three conditions in the example, only the `canFork` variable is checked in the rest of the file [40]. In Listing 5, instead of removing or updating the toggle at all locations in the file, the toggle can be removed or updated in Lines 3 and 4.

```
1 // Set the value of a variable using a feature toggle.
2 const canFork = props.selection.isSingleDocument() &&
3   props.me.feature_toggles &&
4   props.feature_toggles.includes("forking");
```

Listing 5: Use feature toggles to set value to a new variable.

$H4_{NFE2}$: One way to store the value of a toggle is using configuration files, but in some repositories more than one configuration file exists for the same set of feature toggles. The Multiplication Tables Check (MTC) project of UK Department of Education [41] has 14 files for feature toggles. To remove or to edit a feature toggle, a developer must remove or edit the toggle in all of the 14 configuration files. Missing any of these files could cause issues, such as dead

code. The reason for using more than one file could be project-specific, such as managing multiple platforms, but it increases complexity and decreases the maintainability of the code.

4.5. Avoid Duplicate Code in Using Feature Toggles (57)

Heuristic 5 (AvoidDuplicate). When a feature toggle that wraps the same code is used more than once in the same file, *creating a method containing the feature toggle with the wrapped piece of code* can improve maintainability.

Duplicate code is a code smell [42]. When the consequent fragment of code wrapped by a feature toggle's conditional if-statement appears more than once in the same file; that counts as duplicate code, adds complexity to the code, and may create dead code. However, the *extract method* refactoring pattern [42] could be used with feature toggle's consequent fragment of code to avoid duplicate code and subsequent repercussions

H5$_{\mathbf{NFE1}}$: In Salesforce's refocus repository [43], the code in Listing 6 is a fragment of code wrapped with a feature toggle. This block of code appears twice in the same file. The *extract method* refactoring pattern could be used to prevent duplicate code.

```
if (featureToggles.isFeatureEnabled('enableWorkerActivityLogs') &&
    jobResultObj && logObject) {
  mapJobResultsToLogObject(jobResultObj, logObject);
  if (featureToggles.isFeatureEnabled('enableQueueStatsActivityLog'
    )) {
    queueTimeActivityLogs.update(jobResultObj.recordCount,
    jobResultObj.queueTime); }
  activityLogUtil.printActivityLogString(logObject, 'worker');
 }
```

Listing 6: Code block that appears twice in the same file [43].

4.6. Test Cases for Feature Toggles (17)

Heuristic 6 (Testing). Including *test cases* for each feature toggle can improve maintainability.

Software testing is a recommended activity for assessing the quality and correctness of the code [44]. Feature toggles can determine the logic flow and behavior of a product, so must be correct and of high quality. Automated test cases of feature toggles can be used as regressions test which enhances maintainability. Test cases should remain in the code if the development team decides to make the feature permanent.

H6$_{\mathbf{FE1}}$: In the Multiplication Tables Check (MTC) project of UK Department of Education [45], when developers decided to make a feature wrapped in a feature toggle permanent, they removed unit tests for a disabled toggle and kept the tests for an enabled toggle with changed names.

4.7. Complete Removal of a Feature Toggle (21)

Heuristic 7 (CompleteRemoval). Ensuring *complete removal* of a feature toggle by removing it from source code files, configuration files, and test cases can improve maintainability.

Developers should remove feature toggles when the purpose of using the toggles is accomplished. They should remove the code related to the feature toggle from all files in the source code, including configuration files and test files. Incomplete removal can cause problems such as dead code [3]. One solution to ensure complete removal of a feature toggle is to have an implementation that throws a compilation error when a feature toggle exists in the code after being removed from the configuration files.

H7$_{\mathbf{FE1}}$: In this commit [46], the developers of the Multiplication Tables Check (MTC) project from UK Department for Education completely removed a "prepareCheckMessaging" feature toggle from the configuration files, code, and test cases.

5. FT-metrics

We now describe the FT-metrics we identify to support the FT-heuristics. Following the steps in Section 3.2, first, we manually analyze the 60 repositories

in the analysis set. We identify 12 metrics based on our observations of structuring feature toggles in the repositories. Based on their effect on the code base, using card sorting technique and existing categorization in the literature [21], we group these metrics into three categories: *Complexity*, *Comprehensibility*, and *Maintainability*.

Below, we list the 12 FT-metrics. The type of each metric, binary or numeric, is mentioned after the metric's name. For binary metrics (M3–M6, M10–M12), in this study, we consider the presence or absence of the metric in the repositories. A higher level of granularity may be obtained via an automated tool, as discussed in Section 8. FT-metrics are as follows:

Complexity is the degree to which a system has a design or implementation that is difficult to understand and verify [47].

- *M1: Number of added paths in code (Numeric)* is computed using the McCabe's Cyclomatic Complexity [48]. McCabe's Cyclomatic Complexity counts the number of paths in code based on the number of decision points. Incorporating a feature toggle adds decision points to the code, so we can use this metric to compute the added complexity. We focus on the change in the code when developers use feature toggles, so we measure the "change" of the Cyclomatic complexity of the code. For example, if adding a feature toggle adds one `if` statement in the code, we count "+1" for the Cyclomatic complexity of the code. The lower the number of added paths in the code, the better.
- *M2: Number of feature toggle value checking methods (Numeric)* is measured based on the concept behind Weighted Methods per Class (WMC). WMC is one of the CK object-oriented metrics [7] which is computed as the number of methods in a class. To compute this metric, we count the number of feature toggle value checking methods in the feature toggle class manually. We assume all the methods have equal complexities, so the weight for all is 1.0.

Comprehensibility is the degree to which a system is understandable to the developers.

- *M3: Presence of guidelines (Binary)* helps developers know the processes of adding and removing a feature toggle. The absence of guidelines may cause problems such as the creation of dead code after a toggle removal. Guidelines may be provided as a document in repositories or as comments in feature toggles' configuration files.
- *M4: Intention-revealing names (Binary)* for variables and methods is a known practice in coding [31]. A feature toggle's name and related methods should tell the reader what value the toggle holds and what task does the code wrapped by it accomplishes. M4 is a subjective metric.
- *M5: Use of comments (Binary)* as human-readable notes that support the source code is a coding practice [32] that helps developers understand the purpose and behavior of the feature toggles.
- *M6: Use of description (Binary)* for each feature toggle can be used to clarify a toggle's purpose. The description could be added as an attribute to the feature toggle class. Listing 4 in Section 4 is an example of including descriptions. Unlike comments which are not available everywhere, object attributes are accessible throughout the code base.

Maintainability is the ease with which a software system can be modified to correct faults, improve performance or other attributes, or adapt to a changing environment [47].

- *M7: Number of files (Numeric)* which contain a feature toggle, including configuration, code, and test files. The higher the number of files that need to be changed to support feature toggles, the higher the probability to make a mistake. We count the number of files for each feature toggle and then average it for each repository based on the number of toggles. The number of the files could be context-dependent. For instance, separate platforms can have separate configuration files.
- *M8: Number of locations (Numeric)* where a feature toggle is defined and used. As an example, consider a toggle used in two files. In a configuration file, a toggle is mentioned once to set the value of the toggle and, in another

file, the same toggle is used twice in if-statements. In this case, the number of locations for this toggle is three. We count the number of locations where each feature toggle is defined and used and then average the count for each repository.

- *M9: Feature toggle lines of code (Numeric)* which are directly associated with a feature toggle when the toggle is added or removed from a repository. In general, the number of lines of code is a metric to measure maintainability in software systems [49]. In our definition, this metric measures the effort a developer should expend to make any change to the code related to a feature toggle. We count the lines of codes for defining and testing each feature toggle (and not feature toggle usage and enclosed code) and then average it on the number of toggles in each repository.

- *M10: Presence of duplicate code (Binary)* is a code smell [50]. Duplicate code is a problem of repeating the same block of the code. We consider code fragment that contains checking the value of a feature toggle in a conditional statement and the piece of code wrapped by the toggle as duplicate code. In case of updating or removing the toggle, all occurrences of the duplicate code need to be updated or removed.

- *M11: Presence of dead code (Binary)* is one of the drawbacks of using feature toggles in an incorrect way. Dead code is a part of the code which is not used in any execution path [51]. If developers decide to remove a feature toggle, the toggle should be removed from all parts of the code, including configuration files, code, and test cases. In this study, dead code is considered "present" if we found a feature toggle definition or test cases for a toggle but that toggle is not used in the code anymore.

- *M12: Presence of test cases (Binary)* for feature toggles is a metric to measure whether feature toggles are tested. Feature toggles should be tested similarly to implementation code. We consider two types of test cases: (1) checking the values of the feature toggles; and (2) checking the behavior of the code based on the value of the feature toggle. If the repository has any type of test

Table 2: Hypothesized relationship between FT-heuristics and FT-metrics.

Categories	Metrics	H1	H2	H3	H4	H5	H6	H7
Complexity	M1 (Paths)				✔			
	M2 (Methods)	✔						
Comprehensibility	M3 (Guidelines)			✔				
	M4 (Intention)		✔					
	M5 (Comments)		✔					
	M6 (Description)		✔					
Maintainability	M7 (Files)	✔			✔			✔
	M8 (Locations)				✔	✔		
	M9 (LOC)	✔				✔		✔
	M10 (Duplicate)					✔		
	M11 (Dead)	✔		✔	✔			✔
	M12 (Test cases)						✔	

H1 to H7 are list of FT-heuristics: SharedMethod (H1), SelfDescriptive (H2), Guidelines (H3), UseSparingly (H4), AvoidDuplicate (H5), Testing (H6), CompleteRemoval (H7)

cases for the majority of the feature toggles in the code base, we record "yes" for this metric.

Table 2 shows hypothesized relations between FT-heuristics and FT-metrics. The hypothesized relations are determined based on observations in Phase 1 of the methodology and iterative discussions between the first two authors.

6. Survey and Case Study

We now explain Phase 2 of the method in Figure 1 and Section 3.3. We propose the following sub-research questions to investigate RQ_S on evaluating FT-heuristics and FT-metrics.

SRQ_{PA} (Practitioners' agreement): To what extent do practitioners agree that the FT-heuristics can be used to guide practitioners on how to structure and use feature toggles to reduce technical debt?

SRQ_{HM} (Heuristics and metrics): What is the relation between adoption of FT-heuristics and values of FT-metrics?

To address these proposed sub-research questions, we conduct a survey of practitioners from 80 repositories, and a case study with all 80 repositories as the case study set.

6.1. Practitioners Agreement by Survey (SRQ_{PA})

To evaluate the acceptance of FT-heuristics by practitioners, we asked the practitioners the difficulty in managing feature toggles and the extent of their agreement that the FT-heuristics can be used to guide practitioners on how to structure and use feature toggles to reduce technical debt. We used a five-point *Likert* scale for difficulty: Not at all difficult (1) to Very difficult (5); for agreement: Strongly disagree (1) to Strongly agree (5). We included an N/A option too.

To reach out to practitioners and elicit their responses on the survey questionnaire, we first submitted 72 issues in 80 repositories in our dataset. The settings of the other 8 repositories did not allow us to submit an issue. We received 8 responses via issues. Next, we sent 57 emails to practitioners of 45 repositories associated with feature toggles' commits and changed files, and received 12 responses. For 35 repositories, email addresses of feature toggles' contributors were not available. Table 3 shows the 20 survey respondents' experience and frequency of using feature toggles.

The 20 practitioners (survey respondents) have 3–5 years (median) of experience of using feature toggles. They use feature toggles in half of their projects

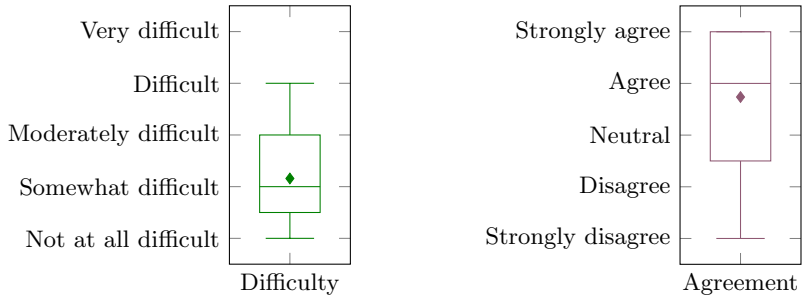
Table 3: Survey respondents' experience and frequency of using feature toggles. Underline indicates median.

Experience	# practitioners	Usage frequency	# practitioners
1–3 years	5	Rarely in projects	6
3–5 years	7	Half of the projects	6
5–8 years	3	Most of the projects	5
8+	0	All of the projects	2
N/A	5	N/A	1

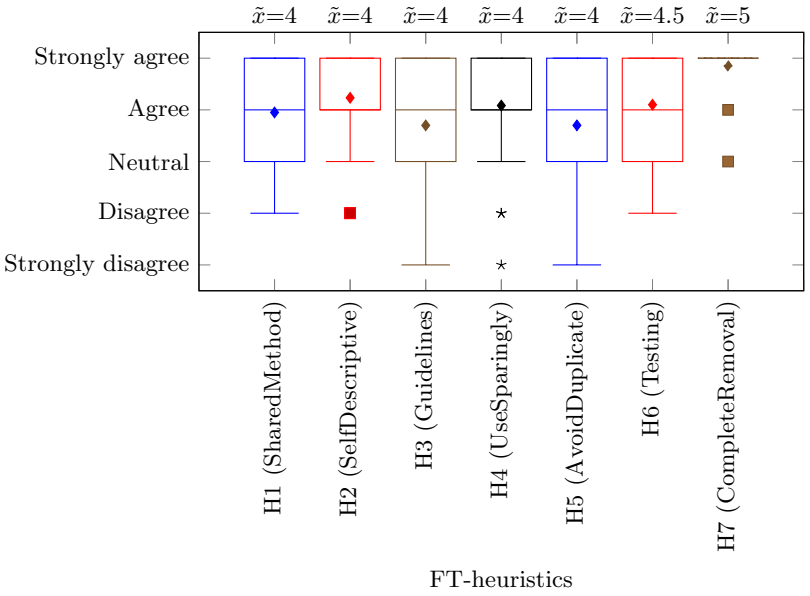
(median). As summarized in Figure 2, the practitioners perceive managing feature toggles to be somewhat difficult and agree that feature toggles increase technical debt. Of the 20 survey respondents, 19 practitioners *strongly agree* and one practitioner *agree*s with CompleteRemoval (H7), stressing the importance of complete removal of feature toggle and how that can improve maintainability. The medians of the agreement to all FT-heuristics are over 4 indicating *Agree*, and the individual means of the agreement to FT-heuristics are all at least 3.7, which is close to *Agree*. The respondents agree that following each of the seven FT-heuristics could guide practitioners on how to structure and use feature toggles to reduce technical debt.

We also asked practitioners for their suggestions to reduce the technical debt of using feature toggles. Only one of the practitioners answered. The practitioner suggested considering the count of developers who interact with each toggle and the last time each changed the code in the path of using the feature toggle. This is similar to the spreadsheet that Chrome's developers use to record the owner of feature toggles [1]. Future works could consider including this metric and corresponding heuristic.

Although we use a survey to evaluate the acceptance of FT-heuristics by practitioners, the main focus of Phase 2 of the methodology (Survey and Case Study) is on the case study to find the relation between the adoption of FT-



(a) Difficulty of feature toggle management. (b) Feature toggle usage increases technical debt.



(c) Extent of agreement for FT-heuristic. $\tilde{x}$ is median; ♦ is mean.

Figure 2: Summary of practitioner survey.

heuristics and the values of FT-metrics.

6.2. Repository Inspection

We inspect each repository in the case study set and identify the toggles in its master branch. From 80 repositories in the case study set, 9 repositories have no feature toggle in their master branch. So 71 repositories are used for case

study analysis. For each toggle, we manually compute FT-metrics, as described in Section 5, and the context metrics, as described in Section 3.3 in the last snapshot of each repository. We also manually identify if each repository follows the FT-heuristics using the following criteria and hypothesized dependent FT-metrics for each heuristic:

SharedMethod (H1): if the values of all toggles are checked using a shared value checking method; not applicable to repositories that check primitive values of toggles in conditional statements.

SelfDescriptive (H2): if the repository has at least two of the following three FT-metrics for the majority of the toggles: (1) intention-revealing names; (2) use of comments; or (3) and use of description.

Guidelines (H3): if the repository has guidelines to manage feature toggles.

UseSparingly (H4): if, based on an expert's (first author's) subjective judgment a feature toggle could be used in fewer files or locations.

AvoidDuplicate (H5): if the repository does not have feature toggle duplicate code based on an expert's (first author) subjective judgment.

Testing (H6): if the feature toggles have associated test cases.

CompleteRemoval (H7): if there is no trace of toggles in the codebase for which there are associated commit message(s) referring to toggle removal.

We also checked the usage of available feature toggle management packages in these repositories. Only 13 of those use feature toggle management packages. In the remaining 67 repositories, the contributors implement their own feature toggle management approach.

6.3. FT-heuristics and FT-metrics (SRQ_{HM})

We now address $\mathrm{SRQ_{HM}}$ on the relation between the adoption of FT-heuristics and the values of the FT-metrics.

Tables 4, 5, and 6 summarize our results. In Table 4, # of repositories, # of contributors, # of feature toggles are context metrics, in Table 5, M1, M2, M7–M9 are numeric FT-metrics, and in Table 6, M3–M6 and M10–M12 are binary FT-metrics. In all three tables 'F' is for repositories that follow a

heuristic and 'NF' is for the repositories which do not follow that heuristic. For numeric metrics, the values for 'F' and 'NF' in Table 5 are the normalized average μ (except for M2 which is an absolute number). For example, the average number of files (M7) for the repositories that follow SharedMethod (H1) is 4.3 versus 4.2 for the repositories that do not follow H1. For binary metrics in Table 6, the values are the fraction of the repositories that have the metric in 'F' repositories or 'NF' repositories. For instance, 40% of the repositories that follow SharedMethod (H1) have test cases (M12) and 20% of the repositories that do not follow H1 have M12.

In Tables 5 and 6, the gray cells show the hypothetical relation between FT-heuristics and FT-metrics. The hypothesized relations are determined based on observations in Phase 1 of the methodology and iterative discussions between the first two authors as shown in Table 2. The results of the case study may support these relations or show new relations.

In the following paragraphs, we analyze the results for each FT-heuristic based on Table 4, Table 5, and Table 6. The percentage improvements are calculated by Equation 1:

$$\text{Improvements} = \frac{\text{F} - \text{NF}}{\text{NF}} \times 100 \tag{1}$$

Improvements can be positive or negative. For example, for the FT-metric Guidelines (M3) in SharedMethod (H1) FT-heuristic, the improvement is 303.9%, i.e., the existence of guidelines in repositories that follow H1 is 303.9% more than those that do not follow H1. For the same FT-heuristic (H1), the improvement value for the number of the value-checking method (M2) is –93.7% that shows the number of the value checking methods in repositories that follow H1 is 93.7% lower, which is reasonable. The improvements are shown in Tables 4, 5, and 6 as Im.

SharedMethod (H1): We observe that repositories which follow H1 ($n = 26$) have more contributors and feature toggles compared to those which do not

Table 4: Observations from case study of 71 repositories for context metrics.

			H1(SharedMethod)	H2(SelfDescriptive)	H3(Guidelines)	H4(UseSparingly)	H5(AvoidDuplicate)	H6(Testing)	H7(CompleteRemoval)
Context Metrics	n (# repositories)	F	26	19	10	53	57	17	21
		NF	45	52	61	18	14	54	13
	# contributors	F	40.3	39.1	73.5	19.7	21.6	24.4	24.6
		NF	20.3	23.4	20.1	50.8	52.2	28.6	53.9
		Im	98.5	67.1	266.0	−61.2	−58.7	−14.7	−54.4
	# feature toggles	F	19.5	20.1	27.7	12.0	10.9	7.0	14.8
		NF	9.5	10.7	10.8	16.8	22.6	15.2	26.2
		Im	105.0	87.5	156.0	−28.6	−52.0	−53.8	−43.5

follow H1 ($n = 45$). In repositories which follow H1, from our hypothesized relationships, number of value checking method (M2) is 93.7% lower; but the number of files (M7), lines of code (M9), and presence of dead code (M11) do not have significant difference compared to repositories not following H1. We unexpectedly observe the metric presence of guidelines (M3) is improved 303.9%, and presence of test cases (M12) is improved 147.3% for repositories which follow H1. As we mentioned 13 repositories use feature toggle management packages. From these 13 repositories, 10 repository have feature toggles in their master branch. From these 10 repositories, 8 repositories follow H1. This shows that having a shared method to check the value of feature toggle is an accepted heuristic for feature toggle management package providers.

SelfDescriptive (H2): We notice that the repositories that follow H2 ($n = 19$) have more contributors and more feature toggles compared to repositories without self-descriptive feature toggles ($n = 52$). From hypothesized relationships,

Table 5: Observations from case study of 71 repositories for numeric metrics. Gray cells indicate the hypothetical relation between FT-heuristics and FT-metrics as mentioned in Table2.

			H1(SharedMethod)	H2(SelfDescriptive)	H3(Guidelines)	H4(UseSparingly)	H5(AvoidDuplicate)	H6(Testing)	H7(CompleteRemoval)
Numeric Metrics	M1 (Paths)	F	1.4	1.7	1.7	1.3	1.3	1.9	2.0
		NF	1.5	1.4	1.4	2.0	2.2	1.4	1.0
		Im	−8.7	19.2	21.0	−33.4	−41.8	38.6	111.4
	M2 (Methods)	F	1.1	9.6	1.0	6.3	5.1	2.9	2.2
		NF	17.0	3.7	6.6	3.1	6.8	7.0	5.6
		Im	−93.7	156.6	−84.8	102.4	−25.1	−58.0	−60.3
	M7 (Files)	F	4.3	4.9	5.2	3.4	4.0	5.0	5.2
		NF	4.2	4.0	4.1	6.9	5.1	4.0	4.8
		Im	3.5	20.7	28.2	−51.1	−20.2	26.3	7.7
	M8 (Locations)	F	5.7	6.0	7.3	5.0	5.5	8.1	7.3
		NF	5.9	5.7	5.6	8.1	7.1	5.1	5.6
		Im	−2.5	4.4	31.6	−37.8	−23.2	60.6	31.7
	M9 (LOC)	F	7.1	5.5	4.7	5.8	5.9	13.7	10.8
		NF	5.7	6.5	6.5	7.4	7.5	3.8	4.8
		Im	24.7	−14.3	−26.6	−21.4	−20.9	257.8	127.4

presence of comments (M5) and presence of descriptions (M6) are higher and improved by more than 3,000% in repositories which follow H2. In addition, we observe that these repositories have 41.4% less duplicate code (M10), and 31.6% less dead code (M11). Since having the intention-revealing names is one of the known coding practices [31], we observe that M4 is the most existed metric in all the repositories.

Table 6: Observations from case study of 71 repositories for binary metrics. Gray cells indicate the hypothetical relation between FT-heuristics and FT-metrics as mentioned in Table2.

			H1(SharedMethod)	H2(SelfDescriptive)	H3(Guidelines)	H4(UseSparingly)	H5(AvoidDuplicate)	H6(Testing)	H7(CompleteRemoval)
Binary Metrics	M3 (Guidelines)	F	0.3	0.2	0.9	0.2	0.1	0.2	0.2
		NF	0.1	0.1	0.0	0.1	0.3	0.1	0.3
		Im	303.9	17.3	5390.0	35.9	−63.2	36.1	−38.1
	M4 (Intention)	F	0.9	1.0	1.0	1.0	1.0	1.0	1.0
		NF	1.0	0.9	1.0	0.9	0.9	0.9	1.0
		Im	−5.6	6.1	5.2	10.4	3.9	5.9	0
	M5 (Comments)	F	0.2	0.6	0.3	0.2	0.1	0.1	0.2
		NF	0.1	0.0	0.1	0.2	0.2	0.2	0.0
		Im	44.2	–	128.8	−9.4	−34.5	−29.4	–
	M6 (Description)	F	0.2	0.6	0.3	0.2	0.2	0.1	0.2
		NF	0.2	0.0	0.2	0.2	0.1	0.2	0.1
		Im	48.4	3184.2	83.0	−23.6	194.7	−73.5	209.5
	M10 (Duplicate)	F	0.2	0.2	0.6	0.2	0.1	0.3	0.3
		NF	0.2	0.3	0.2	0.4	1.0	0.2	0.4
		Im	−5.6	−41.4	232.7	−61.8	−94.7	−32.4	−13.3
	M11 (Dead)	F	0.4	0.3	0.6	0.3	0.4	0.3	0.1
		NF	0.3	0.4	0.3	0.4	0.4	0.4	0.9
		Im	15.4	−31.6	92.6	−12.7	−1.8	−20.6	−89.7
	M12 (Test cases)	F	0.4	0.1	0.4	0.3	0.2	1.0	0.4
		NF	0.2	0.3	0.2	0.2	0.3	0.0	0.3
		Im	147.3	−63.5	87.7	10.4	−20.2	–	23.8

Guidelines (H3): Similar to SharedMethod (H1), repositories that follow H3 ($n = 10$) have a 266% more contributors and 156.0% more feature toggles. The metric presence of guidelines (M3) is hypothesized metric for H3 and it is better and 5,390% more in repositories which follow the heuristic. Other hypothesized metric is dead code (M11) which we observe that it is 92.6% lower in repositories which do not follow H3. Among non-hypothesized relationships, in repositories which follow H3, the metrics presence of duplicate code (M10) is on average 232.7% higher . So, we observe that mere having documented guidelines or using issues or pull requests for structuring feature toggles does not necessarily prevent the occurrence of dead code and duplicate code resulting from feature toggle usage.

UseSparingly (H4): In contrast to SharedMethod (H1) and Guidelines (H3), H4 is followed more in repositories ($n = 53$) with a lower number of contributors and a lower number of the feature toggles. From hypothesized relationships, the repositories which follow H4 have 51.1% lower number of files (M7), and 37.8% lower number of locations (M8). For the rest of the metrics, nothing specific is observed except for the presence of duplicate code (M10) which is 61.8% lower by following H4.

AvoidDuplicate (H5): We observe that repositories which follow H5 ($n = 57$) on average have fewer contributors and a lower number of feature toggles compared to repositories which do not follow H5 ($n = 14$). We note that in repositories which follow H5, from hypothesized relationships, presence of duplicate code (M10) is 94.7% less. In addition, in these repositories, number of paths (M1) is 41.8% less. However, presence of guidelines (M3) are 63.2% lower compared to repositories which do not follow H5.

Testing (H6): We notice that repositories ($n = 17$) with a lower number of feature toggles follow H6. H6 has one hypothesized relationship with FT-metric presence of test cases (M12), which is obviously better in repositories which follow H6. We observe that the number of locations (M8) are 60.6% more and lines of code (M9) are 257.8% more for repositories that follow H6 compared to those that do not follow H6 . Larger values for M8 and M9 are reasonable as

having test cases increase the number of the locations and lines of code related to feature toggles.

CompleteRemoval (H7): We observe that repositories with a larger number of contributors and a larger number of feature toggles do not follow the H7. From hypothesized relationships, repositories that follow H7 have less dead code (M11) by 89.7%. Among non-hypothesized relationships, use of comments (M5) is higher (non of the not following repositories use comments), number of paths (M1) is 111.4% higher, and number of value checking methods (M2) is 60.3% lower .

7. Threats to Validity

Internal validity. We searched GitHub for keyword "feature toggle" and checked only the first 400 search results. We may have missed repositories that use feature toggles in their development process. Developing FT-heuristics and identifying FT-metrics could be subjective to the first author's knowledge. To mitigate this threat, the second author critically reviewed the FT-heuristics and FT-metrics and give feedback. We also conducted a survey to evaluate the findings with practitioners of GitHub repositories in our dataset. Although, FT-heuristics and FT-metrics cover the lifecycle of a feature toggle from design to clean-up and our process was iterative and involved more than one person, we do not claim the completeness of our findings. Future works could find more heuristics and metrics.

In the case study, we only discuss the trends in improvement in FT-metrics when following FT-heuristics. In Appendix A, we report on our findings from preliminary statistical analyses of the case study data. The statistical findings could be strengthened with replication on a larger dataset.

External validity. We use open source repositories from GitHub for our study. Including repositories from other organizations, such as proprietary organizations, may change the results of our study. To check the generalization of our result, we performed a case study on a set of repositories. If more repositories

were analyzed in the case study, we would have stronger evidence of generalization.

Construct validity. Incorrect classification of a code snippet in the 80 GitHub repositories as a feature toggle could render our results invalid. To mitigate this threat, the first author critically examined each feature toggle implementation in consultation with the second author.

8. Lessons Learned and Future Work

Lessons learned.

Our survey respondents—practitioners who routinely use feature toggles, agree with (1) the difficulty of managing feature toggles; (2) the increase of technical debt by using toggles; and (3) the FT-heuristics can be used to guide practitioners on how to structure and use feature toggles to reduce technical debt.

Based on our case study and survey observations, we note:

SharedMethod (H1). Using *shared method* is more common in repositories with guidelines and test cases compared to those without guidelines and test cases. However, we did not observe any meaningful difference in number of files, lines of code, and dead code when following this heuristic compared to not following it.

SelfDescriptive (H2). Having *self-descriptive feature toggles* help in preventing duplicate code and dead code in a repository. We did not hypothesize the relation between self-descriptive feature toggles and and metrics duplicate code and dead code but we find these metrics to be lower with repositories having self-descriptive feature toggles.

Guidelines (H3). Providing *guidelines* to manage feature toggles may not necessarily reduce duplicate and dead code. Practitioners may mandate guidelines by other means such as code review.

UseSparingly (H4). Using *toggles sparingly* is the second most followed FT-

heuristic in our case study set. Doing so reduces duplicate code, as a non-hypothesized metric.

AvoidDuplicate (H5). Avoiding *duplicate code* in using feature toggles is the most followed FT-heuristic in our case study set. This shows that developers are aware of the negative effects of having duplicate code for feature toggles as other parts of the code.

Testing (H6). Although surveyed practitioners agree on having *Test cases* for feature toggles, this heuristic is second least followed in our case study. Finding the effect of the lack of test cases for feature toggles is a direction for future studies.

CompleteRemoval (H7). *Complete removal* of a feature toggle received the highest agreement by survey respondents. Following FT-heuristic on ensuring complete removal of feature toggles reduces the presence of the dead code.

As a result, we suggest practitioners create self-descriptive feature toggles (H2), use feature toggles sparingly (H4), avoid duplicate code in using feature toggles (H5), and ensure complete removal of a feature toggle (H7). Practitioners may follow FT-heuristics without measuring FT-metrics. However, we strongly suggest practitioners to consider four FT-metrics: number of feature toggle value checking methods (M2), presence of guidelines (M3), presence of duplicate code (M10), and presence of test cases (M12).

Since the identified FT-heuristics are very similar to general software engineering best practices in software development literature, feature toggles should be considered as regular code even though they will not reside permanently in the code. However, even these general software engineering best practices are not followed by all of the analyzed repositories. Not structuring feature toggles correctly, can cause severe damages to the project. Thus, increasing the awareness of the effect of following and not following FT-heuristics is important.

Future work. The FT-metrics can be validated by 47 criteria to validate software metrics extracted by Meneely et al. [52] in future work. In our study, we manually computed the metrics. An automated tool for computing FT-metrics and applying FT-heuristics for structuring feature toggles in the code base is

a future direction. Such a tool could be used to conduct a larger scale evaluation effort to generalize our findings outside of open source repositories. In this study, we analyze the last snapshot of repositories. A direction for future work is to consider metrics related to "lifetime" of feature toggles from the history of repositories, such as how long the toggle lived in the code base, or who is the last developer who touched the toggle. Considering additional metrics can result in additional heuristics about structuring feature toggles in code bases. Our case study result shows that Testing (H6) is not followed by a large number of repositories (54). We focus on raising awareness about having test cases for feature toggles. Considering the importance of testing strategies and especially combinatorial testing related to configuration options [53], research on combinatorial testing for feature toggles can be a future direction. We conduct preliminary statistical analysis (Appendix A) on the case study dataset and report our findings. A future direction is to conduct a larger-scale empirical study with rigorous statistical analysis to strength these findings on the relationship between following FT-heuristics and improving FT-metrics and finding new relationships.

References

[1] M. T. Rahman, L.-P. Querel, P. C. Rigby, B. Adams, Feature toggles: Practitioner practices and a case study, in: Proceedings of the 13th International Conference on Mining Software Repositories (MSR), ACM, Austin, 2016, pp. 201–211.

[2] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, et al., The top 10 adages in continuous deployment, IEEE Software 34 (3) (2017) 86–95.

[3] Knightmare: A devops cautionary tale, [Online]. Available: `https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/` Accessed 24 April 2019.

[4] E. Tom, A. Aurum, R. Vidgen, An exploration of technical debt, Journal of Systems and Software 86 (6) (2013) 1498–1516.

[5] J. Bird, Feature toggles are one of the worst kinds of technical debt, [Online]. Available: https://dzone.com/articles/feature-toggles-are-one-worst Accessed 6 August 2020.

[6] G. A. Moore, Crossing the Chasm: Marketing and Selling Technology Project, Harper Collins, New York, 2009.

[7] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering (TSE) 20 (6) (1994) 476–493.

[8] J. Liebig, S. Apel, C. Lengauer, C. Kästner, M. Schulze, An analysis of the variability in forty preprocessor-based software product lines, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, 2010, pp. 105–114.

[9] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Pearson Education, Boston, 2010.

[10] M. Fowler, Continuous delivery, [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html, Accessed 6 August 2019, (2013).

[11] A. A. U. Rahman, E. Helms, L. Williams, C. Parnin, Synthesizing continuous deployment practices used in software development, in: Proceedings of the IEEE Agile Conference, IEEE, 2015, pp. 1–10.

[12] R. Harmes, Flipping out, [Online]. Available: http://code.flickr.net/2009/12/02/flipping-out/, Accessed 6 August 2019 (2009).

[13] R. Mahdavi-Hezaveh, J. Dremann, L. Williams, Software development with feature toggles: practices used by practitioners, Empirical Software Engineering 26 (1) (2021) 1–33.

[14] P. Hodgson, Feature toggles (aka feature flags), [Online]. Available: https://martinfowler.com/articles/feature-toggles.html, Accessed 6 August 2019 (2017).

[15] B. Hodges, Progressive experimentation with feature flags, [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/progressive-experimentation-feature-flags, Accessed 6 August 2019.

[16] R. Kohavi, R. Longbotham, D. Sommerfield, R. M. Henne, Controlled experiments on the web: Survey and practical guide, Data mining and knowledge discovery 18 (1) (2009) 140–181.

[17] M. T. Rahman, P. C. Rigby, E. Shihab, The modular and feature toggle architectures of Google Chrome, Empirical Software Engineering (EMSE) 22 (2) (2018) 1–28.

[18] M. K. Ramanathan, L. Clapp, R. Barik, M. Sridharan, Piranha: Reducing feature flag debt at uber, in: Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE), ACM, Seoul, 2020, pp. 1–10.

[19] J. Meinicke, J. Hoyos, B. Vasilescu, C. Kästner, Capture the feature flag: Detecting feature flags in open-source, in: Proceedings of the 17th International Conference on Mining Software Repositories (MSR), Seoul, 2020, pp. 169–173.

[20] J. Meinicke, C.-P. Wong, B. Vasilescu, C. Kästner, Exploring differences and commonalities between feature flags and configuration options, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 233242. doi:10.1145/3377813.3381366.
URL https://doi.org/10.1145/3377813.3381366

[21] M. Sayagh, N. Kerzazi, B. Adams, F. Petrillo, Software configuration engineering in practice: Interviews, survey, and systematic literature review, IEEE Transactions on Software Engineering (TSE).

[22] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, G. Saake, On essential configuration complexity: Measuring interactions in highly-configurable systems, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 483–494.

[23] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, T. Xu, An evolutionary study of configuration design and implementation in cloud systems, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 188–200. doi:10.1109/ICSE43902.2021.00029.

[24] J. Saldaña, The coding manual for qualitative researchers, Sage, 2015.

[25] G. Rugg, P. McGeorge, The sorting techniques: A tutorial paper on card sorts, picture sorts and item sorts, Expert Systems 22 (3) (2005) 94–107.

[26] R. Likert, A technique for the measurement of attitudes, Archives of Psychology.

[27] A. Danial, Cloc: Count lines of code, [Online]. Available: http://cloc.sourceforge.net/, Accessed 24 July 2019.

[28] G. James, D. Witten, T. Hastie, R. Tibshirani, An introduction to statistical learning, Vol. 112, Springer, 2013.

[29] NAV - The Norwegian Labour and Welfare Directorate, pleiepengesoknad, [Online]. Available: https://github.com/FeatureToggleStudy/pleiepengesoknad/blob/master/src/app/utils/featureToggleUtils.ts#L7 Accessed 27 February 2020.

[30] Education and Skills Funding Agency Team, Cfs-backend, [Online]. Available: https://github.com/SkillsFundingAgency/CFS-Backend/blob/d4461bda36e3d785909350233f833594984823c3/Debugging/

CalculateFunding.DebugAllocationModel/FeatureToggles.cs Accessed 21 October 2019.

[31] N. B. Dale, C. Weems, M. R. Headington, Introduction to Java and Software Design: Swing Update, Jones & Bartlett Learning, 2003.

[32] G. Penny, A. A. Takang, Software Maintenance: Concepts and Practice, World Scientific, 2003.

[33] M. Sayagh, N. Kerzazi, F. Petrillo, K. Bennani, B. Adams, What should your run-time configuration framework do to help developers?, Empirical Software Engineering 25 (2) (2020) 1259–1293.

[34] UK Education and Skills Funding Agency Team, Cfs-frontend, [Online]. Available: https://github.com/featuretogglestudy/CFS-Frontend/blob/39961217b8aabd665c71b108903d87014a41582c/DevOps/frontend-azure.dfe.json#L237 Accessed 27 February 2020.

[35] Automattic, wp-calypso, [Online]. Available: https://github.com/Automattic/wp-calypso/blob/8d1bf0b0146fc341288059c765e8a3bf8c8bb7ef/client/me/purchases/manage-purchase/purchase-meta.jsx Accessed 21 October 2019.

[36] Salesforce, refocus, [Online]. Available: https://github.com/salesforce/refocus/blob/18116cb7df73c0db70af3c6115342ccf93db6534/config/toggles.js Accessed 21 October 2019.

[37] UK HM Courts & Tribunals Service , div-case-orchestration-service, [Online]. Available: https://github.com/hmcts/div-case-orchestration-service Accessed 24 August 2020.

[38] The Guardian, Grid, [Online]. Available: https://github.com/guardian/grid Accessed 27 February 2020.

[39] Australian Department of Veterans' Affairs, myservice-prototype, [Online]. Available: https://github.com/AusDVA/myservice-prototype Accessed 9 October 2019.

[40] Pelagios Network, recogito2-workspace-frontend, [Online]. Available: https://github.com/pelagios/recogito2-workspace-frontend/blob/367723732cfa74c4f38e542b88c0a4491789cc04/src/profile/Profile.jsx Accessed 9 October 2019.

[41] UK Department for Education, Multiplication Tables Check (MTC) Project, [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/tree/0eb2d765b6683c90c852ba21c225742f07f050b9/admin/config Accessed 9 October 2019.

[42] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[43] Salesforce, refocus, [Online]. Available: https://github.com/salesforce/refocus/blob/18116cb7df73c0db70af3c6115342ccf93db6534/jobQueue/jobWrapper.js Accessed 9 October 2019.

[44] M. A. Ould, C. Unwin, Testing in software development, Cambridge University Press, 1986.

[45] UK Department for Education, Multiplication Tables Check (MTC) Project, [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/commit/0eb2d765b6683c90c852ba21c225742f07f050b9#diff-8633026cf78840f2cb5a5b32fe1aa00f Accessed 11 October 2019.

[46] UK Department for Education, Multiplication Tables Check (MTC) Project, [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/commit/0eb2d765b6683c90c852ba21c225742f07f050b9 Accessed 27 February 2020.

[47] IEEE Standards Coordinating Committee, IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos, CA: IEEE Computer Society 169.

[48] T. J. McCabe, A complexity measure, IEEE Transactions on Software Engineering (TSE) 2 (4) (1976) 308–320.

[49] I. Heitlager, T. Kuipers, J. Visser, A practical model for measuring maintainability, in: Proceedings of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC), IEEE, 2007, pp. 30–39.

[50] C. K. Roy, J. R. Cordy, A survey on software clone detection research, Queen's School of Computing TR 541 (115) (2007) 64–68.

[51] H. Xi, Dead code elimination through dependent types, in: G. Gupta (Ed.), Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings, Vol. 1551 of Lecture Notes in Computer Science, Springer, 1999, pp. 228–242. doi:10.1007/3-540-49201-1_16.
URL https://doi.org/10.1007/3-540-49201-1_16

[52] A. Meneely, B. Smith, L. Williams, Validating software metrics: A spectrum of philosophies, ACM Transactions on Software Engineering and Methodology (TOSEM) 21 (4) (2013) 1–28.

[53] M. B. Cohen, J. Snyder, G. Rothermel, Testing across configurations: implications for combinatorial testing, ACM SIGSOFT Software Engineering Notes 31 (6) (2006) 1–9.

Appendix A. Preliminary Statistical Analysis

In this section, we report on the preliminary statistical analysis we conduct on the case study dataset.

Algorithm 1 Best Subset Selection algorithm adapted from [28]

For each FT-metric:

For $k = 1, 2, \ldots, p$: # Identify the best model for each k

if FT-metric is numeric **then**

(1) Fit all $\binom{p}{k}$ linear regressions that contain exactly k predictors

(2) Select μ_k = Model with largest adjusted R^2 (best model)

else if FT-metric is binary **then**

(1) Fit all $\binom{p}{k}$ logistic regressions that contain exactly k predictors

(2) Select μ_k = Model with lowest LLR p-value

end if

Select a single best model from $\mu_1, \ldots, \mu_p$

if FT-metric is numeric **then**

(1) Conduct ANOVA to compare the models

(2) Select $\text{bestmodel}_{\text{metric}}$ = Model with lowest F-test p-value where $p < 0.05$

else if FT-metric is binary **then**

(1) Compare LLR p-values of the models

(2) Select $\text{bestmodel}_{\text{metric}}$ = Model with lowest LLR p-value where $p < 0.05$

end if

We perform *Best Subset Selection* [28] and identify the best subset of predictors (including context metrics and FT-heuristics) to improve each one of the FT-metrics. Algorithm 1 outlines this process to identify the best subset for each FT-metrics. Since FT-heuristics are categorical predictors, we have created dummy variables for them [28]. So, we have 17 predictors to use in our regression analysis. Following Algorithm 1, for each FT-metric, we fit 17 models with 1 predictor, 136 models with 2 predictor, 680 models with 3 predictor, 2,380 models with 4 predictor, 6,188 models with 5 predictor, 12,376 models with 6 predictor, 19,448 models with 7 predictor, 24,310 models with 8 predictor, 24,310 models with 9 predictor, 19,448 models with 10 predictor, 12,376 models with 11 predictor, 6,188 models with 12 predictor, 2,380 models with 13 predictor, 680 models with 14 predictor, 136 models with 15 predictor, 17 models with 16 predictor, and 1 model with 17 predictors. In the set of models for any number of predictors, we select the best one. Then from 17 selected models, we select a single *best* model as the overall best model.

For each FT-metric, we report the predictors of overall best model in Table A.7. In Table A.7, $\uparrow$ indicates the predictor in the best subset with positive coefficient, and $\downarrow$ indicates the predictor in the best subset with negative coefficient.

Comparing the result in Table A.7 and Table 2 confirms the correctness of six of the hypothesized relations. For example, we observed that repositories that do not follow H4 (UseSparingly) have higher number of the locations for feature toggles (M8 (Location)) and, repositories that do not follow H7 (CompleteRemoval) of feature toggles have more dead code (M11 (Dead)). In addition, we find 26 new relations (non-hypothesised) between FT-metrics and FT-heuristics. For example, we find a relationship between H1 (SharedMethod) and M10 (Duplicate), meaning that in our case study set when repositories do not have shared method, they have more duplicate code. To strengthen these newly identified relations, future works could conduct a larger-scale empirical study.

Table A.7: Best Subset Selection result for FT-metrics. ↑ indicates the predictor is in the overall best model with positive coefficient. ↓ indicates the predictor is in the overall best model with negative coefficient.

	Metrics	#contributors	#feature toggles	H1(SharedMethod-F)	H1(SharedMethod-NF)	H2(SelfDescriptive-F)	H2(SelfDescriptive-NF)	H3(Guidelines-F)	H3(Guidelines-NF)	H4(UseSparingly-F)	H4(UseSparingly-NF)	H5(AvoidDuplicate-F)	H5(AvoidDuplicate-NF)	H6(Testing-F)	H6(Testing-NF)	H7(CompleteRemoval-F)	H7(CompleteRemoval-NF)	H7(CompleteRemoval-Unknown)
Complexity	M1 (Paths)																	
	M2 (Methods)	↑							↑									
Comprehensibility	M3 (Guidelines)	↓	↑						↓									
	M4 (Intention)	↑	↑	↓	↑					↑	↓					↑		
	M5 (Comments)					↑	↓											
	M6 (Description)		↑	↑			↓	↑	↓		↓		↓		↑		↓	
Maintainability	M7 (Files)																	
	M8 (Locations)										↑			↑				
	M9 (LOC)														↓	↑		
	M10 (Duplicate)	↑			↑								↑			↓		↓
	M11 (Dead)																↑	
	M12 (Test cases)		↓	↑	↓	↓	↑											↓