Exploring Influence of Feature Toggles on Code Complexity

Tajmilur Rahman
Gannon University
Erie, PA, USA
rahman001@gannon.edu

Imran Shalabi
Gannon University
Erie, PA, USA
shalabi001@gannon.edu

Tushar Sharma
Dollhouse University
Halifax, NS, Canada
tushar@dal.ca

ABSTRACT

Feature toggles are conditional variables that control program execution flow. Toggles are used to control feature states and allow developers to introduce unfinished features to a limited user group while maintaining regular software functionality. Due to the lack of comprehensive best practices, guidelines, or a coding standard for using feature toggles, developers often use them in an inappropriate way leading to code quality issues. In this paper, we investigate four feature toggle usage patterns identified in two popular open-source software projects and assess their impact on code-complexity and size. We develop a tool *ts-detector* to identify the usage patterns automatically. Our investigation indicates that spread toggle and mixed toggle usage patterns occur most and least in the analyzed subject systems. We also found that feature toggle usage patterns collectively have a strong influence on the code complexity and size metrics. Our fine-grained analysis reveals that spread and nested toggle usage patterns have a significant correlation with selective size and complexity metrics. This paper not only offers a tool for software developers and researchers to identify the usage patterns and take corrective actions, but also, the study will motivate other researchers to further extend the experiments to understand and mitigate issues arising from misusing feature toggles.

KEYWORDS

Feature Toggle, Toggle Usage Pattern, Toggle Smell, Static Analysis, Mixed Method

ACM Reference Format:
Tajmilur Rahman, Imran Shalabi, and Tushar Sharma. 2018. Exploring Influence of Feature Toggles on Code Complexity. In *Proceedings of The 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Developers use *feature toggles* (also known as flags, switches, and flippers) in conditional statements allowing them to enable or disable features [6]. Primarily, feature toggles are used for feature branching and merging in Continuous Integration and Deployment (CI/CD) environments [13]. Toggles, short for feature toggles, come with benefits and risks. Implementing feature toggles offers the benefit of streamlined merging and simplified deployment processes. However, it also comes with significant responsibilities such as deleting unused toggles and awareness of toggle life span since feature toggles typically have a longer life span [17].

Due to the lack of comprehensive best practices, guidelines, or a coding standard for using feature toggles, developers use them similar to "if" conditions. With the increasing adoption of feature toggles by software companies, the downsides of not properly maintaining feature toggles have surfaced as unfortunate tales [21]. This indicates the importance of identifying and categorizing potential improper usages of feature toggles. Similar to code smells [7, 22], improper usages of toggles may lead to *toggle smells*. Therefore, thorough investigations become necessary not only to understand different feature toggle usage patterns, but also to identify when a certain feature toggle pattern starts to influence aspects of software maintainability (such as readability and understandability).

There has been studies to catalog feature toggles, their usage patterns, and identify them in a large software project. For example, Rahman [18] observed six toggle usage patterns in Google Chromium and reported three empirically. However, overall, the current research do not investigate whether a toggle usage pattern impacts software maintainability and can be identified as toggle smell. Toggle type and toggle usage patterns are different in the context. There are different types of feature toggles, such as, release toggle, dev toggle, business toggle [12, 17]. In this study we are interested in toggle usage patterns regardless of toggle type.

Our aim is to implement algorithms for identifying various toggle usage patterns, quantify their presence in multiple C++ projects, and find correlations with code quality metrics. Towards the goal, we developed a tool, *ts-detector*, to detect four toggle smells observed by Rahman [18] since no tool has been developed to detect feature toggles in the previous study.

The tool that we developed detects *dead toggle*, *nested toggles*, *spread toggle*, and *mixed toggle* usage patterns in C++ based software projects. The tool offers a customizable way of using the tool allowing analysis of software projects with diverse organization and source code structure. We obtain the detected toggles using *ts-detector* empirically and investigate the code complexity of the files containing toggle usages. We compute correlation between toggle usage patterns and code complexity to explore any relationship between the two concepts. Furthermore, we conduct a developers' survey to gather their opinions and compare with our empirically observed results.

2 BACKGROUND AND RELATED WORK

Martin Fowler [6] and Jez Humble emphasized the significance of feature toggles in continuous deployment and feature development. Large software companies, such as Google [24], Facebook [23],

EASE 2024, 18–21 June, 2024, Salerno, Italy

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Uber [2], Netflix [16], Flickr [5], and Apptimize [1], use feature toggles to roll out features in a controlled manner.

Hezaveh *et al.* [13] elucidated the utilization of feature toggles in the software development through qualitative analysis, classifying them into five distinct types: *release toggles, experiment toggles, ops toggles, permission toggles*, and *development toggles*. Their study offered comprehensive insights into operational techniques and diverse strategies for managing feature toggles. Similarly, Rahman *et al.* [17] categorized feature toggles based on their life span and their purpose of use into three types *development toggles, release toggles*, and *business toggles*.

Usage patterns of the C/C++ pre-processors *i.e., "#ifdef"*s have been studied by many researchers [4, 9, 10, 14]. Medeiros *et al.* [14], and Leibig *et al.* [10] discussed the disciplines of using *#ifdef*s. Meinicke *et al.* [15] explored the differences and commonalities between feature toggles (referred to as flags) and C pre-processors. Our study focuses on the usage patterns of feature toggles where one of the usage patterns named as *mixed usage pattern* involves both pre-processors and run-time toggle variable.

Rahman [18] identified six usage patterns of feature toggles in Google Chromium and empirically reported their occurrence characteristics such as frequency and components with multiple toggle usages. Rahman also validated the findings with Google Chromium developers confirming the naming of the usage patterns, and whether developers believe there is a necessity of establishing global standards for using feature toggles. According to the study, Chromium developers express the necessity of having a common standard and guideline of feature toggle usage patterns and believe that there might be some usage patterns with potentials of turning into *toggle smells*. In this context, we would like to extend the aforementioned study to design the algorithms to detect toggle usage patterns, implement them, and make the tool available to the community.

Regular expressions are typically used in identifying feature toggle usage patterns [18]. Though Rahman [18] shared the data used in that study, the regular expressions used to find usage patterns were not shared. Therefore, we designed our own regular expressions for toggle usage pattern identification. Moreover, we made them customizable via a configuration file so that the developed tool can be used on any C++-based code-base. Within the scope of this short paper, we tested *ts-detector* on Google Chromium[1] and Dawn[2] open-source projects.

3 METHODS

3.1 Overview

Our overarching goal is to identify feature toggle usage patterns and explore their effects on code quality. To steer this study and substantiate our analysis, we sought answers to the following research questions.

RQ1. *What feature toggle usages are prevalent in C++ projects?*
With this research question we would like to understand whether certain toggle usage patterns are more common than the others across C++ projects.

[1]https://github.com/chromium/chromium
[2]https://dawn.googlesource.com/dawn

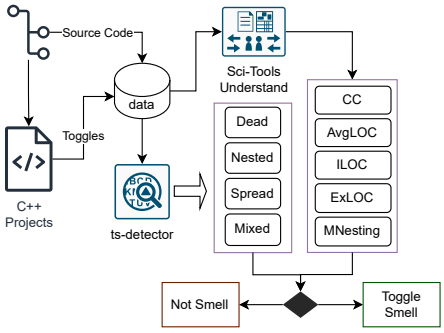

Figure 1: Methodology at a Glance.

RQ2. *Do feature toggle usage patterns collectively exhibit any influence on the code complexity?*
Answering this question will help us identify the role of toggle usage patterns in code complexity.

RQ3. *Do individual feature toggle usage patterns show correlation with code complexity?*
This research question aims to find out whether specific toggles influence code complexity with their presence.

Figure 1 offers an overview of the employed methodology of the study. First, we develop a tool to detect toggle usage patterns supporting detection of *dead toggle*, *nested toggle*, *spread toggle*, and *mixed toggle*. We design the detection algorithm and implement in Python to detect and report from C++ projects. We also obtain code complexity metrics, average Cyclomatic Complexity (CC), and Maximum Nesting (MNesting), and code size metrics, average Lines of Code (AvgLOC), Inactive Lines of Code (ILOC), and Executable Lines of Code (ExLOC) with the help of static analysis tool *Understand* [20].

3.2 Mining Toggles From C++ Projects

To identify the toggle usage patterns, we follow the process outlined by Rahman [18]. First, we collect all the feature toggle variables used in the code. Toggles are typically maintained in configuration files following a specific naming convention. Following code snippet shows some of the feature toggles used to control GPU features specified in a toggle configuration file *gpu_switches.cc* of Chromium.

```
// Disable the GL error log limit.
const char kDisableGLErrorLimit[] = "disable-gl-error-
    limit";
// Disable the GLSL translator.
const char kDisableGLSLTranslator[] = "disable-glsl-
    translator";
// Turn off user-defined name hashing in shaders.
const char kDisableShaderNameHashing[] = "disable-shader-
    name-hashing";
```

Listing 1: Chromium: GPU Toggles

Similarly, we list a couple of feature toggles specified in a configuration file *Features.cpp* of Dawn project.

```
static constexpr FeatureEnumAndInfo kFeatureInfo[] = {
    {Feature::HostMappedPointer,
```

```
    {"Support creation of buffers from host-mapped
    pointers.",
     "https://dawn.../docs/dawn/features/
    host_mapped_pointer.md",
     FeatureInfo::FeatureState::Experimental}},
   {Feature::FramebufferFetch,
    {"Support loading the current framebuffer value in
    fragment shaders.",
     "https://dawn.../docs/dawn/features/
    framebuffer_fetch.md",
     FeatureInfo::FeatureState::Experimental}},
};
```

Listing 2: Dawn: Toggles

The project specific toggles not only are limited to the specific way of naming them and declaring them in their own way; the structure of toggle variable specification is also tend to be project-specific. For example, in Chromium each component has its own set of feature toggles configured in a separate file with a similar naming convention of **_switches.cc*. On the other hand, Dawn lists all feature toggle variables in one single file.

3.3 Toggle Usages

To identify toggle usage patterns, we rely on toggle usage definitions discussed by Rahman [17]. However, we refine and extend Rahman's algorithms to identify toggle usages in a reliable and generalizable manner. We present below the algorithm used for each of the toggle usage patterns considered in this study.

3.3.1 Nested Toggle. It is also referred to as "interaction-between" toggles [25]. This pattern is identified when one feature toggle is dependent on the state of one or more other toggles. Listing 1 presents the employed algorithm to identify nested toggles from C++ projects.

Algorithm 1 Extract Nested Toggle Usage

Require: $sourceFiles, tConfigFiles$
  $nestedToggles \leftarrow newdictionary()$
  $sContents \leftarrow getSourceContents(lang, sourceFiles)$
  $nestedPatterns \leftarrow getNestedTogglePatterns(lang)$
  $distinctNestedToggles = newset()$
  for $sourceFile, contentinsourceFiles, sContents$ **do**
    for $patterninnestedPatterns$ **do**
      $pMatches \leftarrow patternMatch(pattern, content)$
      for $matchinpatternMatches$ **do**
        $nestedToggles[codeFile] \leftarrow match$
      end for
    end for
    $output \leftarrow \{nestedToggles, foundInPaths, countToggles\}$
    $outputJSON \leftarrow json.convert(output)$
    $returnoutputJSON$
  end for

The algorithm requires two parameters: source code path and toggle configuration file name. A toggle is considered in a nested usage when one or more toggle variables are dependent in a nested relationship. In the algorithm, the variable *nestedPatterns* contains the patterns of a typical usage of a toggle variable in nested way.

3.3.2 Spread Toggle. According to Rahman [18], if a feature toggle is used in multiple files and components then this type of spreading nature of usage is referred to as spread toggles.

The prerequisite to identify this usage pattern is to know the list of components for the project. In his study, Rahman provided the list of Chromium components as an input and hence it was easier to ignore the challenges to obtain component information for a software project reliably. However, it could be challenging for a user to understand how the software is structured and what source code elements are actually considered components. In this context, we apply a different strategy to define spread toggles. We identify a feature toggle if a toggle is used in code files more than one folder. Listing 2 shows the key steps in the underlying logic that spots the spread toggles.

Algorithm 2 Extract Spread Toggle Usage

Require: $lang, sourceFiles, tConfigFiles$
  $toggleLookup \leftarrow newdictionary()$
  $spreadToggles \leftarrow newdictionary()$
  $allToggles \leftarrow getExistingToggles(lang, configFiles)$
  $toggleLookup \leftarrow getByDir(lang, sourceFiles, allToggles)$
  $spreadToggles \leftarrow findSpread(toogleLookup)$
  $output \leftarrow \{spreadToggles, countToggles\}$
  $outputJSON \leftarrow json.convert(output)$
  $returnoutputJSON$

3.3.3 Mixed Toggle. This usage pattern is identified when a run-time feature toggle is found within a compile-time macro (#ifdef, #endif) [18]. Listing 3 shows our mechanism to identify mixed toggles.

Algorithm 3 Extract Mixed Toggle Usage

Require: $lang, sourceFiles, tConfigFiles$
  $mixedToggles \leftarrow newdictionary()$
  $sContents \leftarrow getSourceContents(lang, sourceFiles)$
  $mixedPatterns \leftarrow getMixedTogglePatterns(lang)$
  for $sourceFile, contentinsourceFiles, sContents$ **do**
    for $patterninmixedPatterns$ **do**
      $pMatches \leftarrow patternMatch(pattern, content)$
      for $matchinpatternMatches$ **do**
        $mixedToggles[codeFile] \leftarrow match$
      end for
    end for
    $output \leftarrow \{mixedToggles, countToggles\}$
    $outputJSON \leftarrow json.convert(output)$
    $returnoutputJSON$
  end for

3.3.4 Dead Toggle. Feature toggle that no longer exists in the configuration file but has not been removed from where it was used, and actively encapsulating a feature code is referred to as a dead toggle [18]. Listing 4 shows the underlying logic for extracting the dead usage patterns. If the toggle variable is absent in the configuration file(s) but the toggle usages are still present in the source files then such usage is detected as dead toggle usage.

Algorithm 4 Extract Dead Toggle Usage

```
Require: lang, sourceFiles, tConfigFiles
  deadToggles ← newdictionary()
  allToggles ← getExistingToggles(lang, tConfigFiles)
  sContents ← getSourceContents(lang, sourceFiles)
  generalUsagePatterns ← getGeneralUsagePatterns(lang)
  for sourceFile, contentinsourceFiles, sContents do
    for patterningeneralUsagePatterns do
      pMatches ← patternMatch(pattern, content)
      for matchinpatternMatches do
        if matchnotinallToggles then
          deadToggles[codeFile] ← match
        end if
      end for
    end for
    output ← {deadToggles, countToggles}
    outputJSON ← json.convert(output)
    returnoutputJSON
  end for
```

3.4 Tool Support

We implement the above algorithms in a python-based tool. The tool can be downloaded from an anonymous link[3]; the GitHub repository is not provided to adhere to double-blind norms and will be made public if the paper is accepted.

The tool uses regular expressions extensively to identify and report supported feature toggle usage patterns. We define a default set of regular expressions that can be overridden, if required, to customize the functionality of the tool based on the project under analysis. The tool takes optionally a configuration file as an input to help the user specify all project-specific customizations.

The tool can be used by downloading the source code of the tool and running `tsd.py` file using Python environment. An example of the invocation of the tool is provided below.

```
python3 tsd.py <Language> </Source/code/path/> <
    confi_file> <toggle_usage>
```

Listing 3: Command template to use ts-detector

Here, there are four parameters needed from the user, language, source path, configuration file, and toggle usage pattern. For example, if we are running this tool on Chromium source code, the command should look like as follows.

```
python3 tsd.py C++ ~/Downloads/chromium/ switches.cc
    spread
```

Listing 4: Example command to use ts-detector

Although, the tool is currently compatible with only C++ projects, since we are continuing our study on implementing other toggle usage patterns and also to generalize this tool with all possible high-level programming languages, we have the "language" as a parameter.

4 ANALYSIS

To analyze the influence of toggle usage patterns on code complexity we collected two datasets. The first dataset was collected using Understand API [19]. The second dataset is collected by running our ts-detector tool on Chromium and Dawn source code where we collected toggle usages counts per file.

[3]https://shorturl.at/hAEL9

4.1 Complexity Metrics

The powerful python-based API provided by Scitools for the popular static analysis tool *Understand* allowed us to conveniently extract the code quality metrics—average Cyclomatic Complexity per file (CC), average lines of code in methods per file (AvgLOC), number of inactive lines per file (ILOC), Executable LOC per file (ExLOC), and Maximum Nesting level of control constructs per file (MNesting). The code metrics are also collected at the file level including both class files, and non-class files.

Table 1 presents code quality metrics obtained from Chromium and Dawn projects. In the median case, each file in Chromium has an average cyclomatic complexity of 1 with a maximum of 70; on the other hand, Dawn has a similar median complexity with a maximum of 205. Although the Chromium project is larger than Dawn in terms of code size (25GB and 0.5GB of disk space, respectively), each file in Chromium has 8 AvgLOC, while Dawn has 10 in the median case. Clearly, Dawn's files are much larger since it has 1.17 times more Executable LOC (ExLOC) than Chromium. However, the median MNesting (maximum nesting) in all Dawn files is 1, while Chromium has 2.

Table 1: Chromium and Dawn Code-Metrics

	CC	AvgLOC	ILOC	ExLOC	MNesting
Chromium					
Median	1	8	0	12	1
Max	70	682	3563	1703	10
Dawn					
Median	1	10	0	8	0
Max	205	4572	577	2006	8

Many code-metrics have been studied as complexity metrics in the literature [3, 8, 11, 26] while cyclomatic complexity has always been popular among both researchers and practitioners. However, we have included AvgLOC metrics as well along with maximum nesting (MNesting) metrics since toggles may cause nesting, create additional lines of code, leave dead code, and may contribute to the increase of cyclomatic complexity.

4.2 Quantify Toggle Usages

To quantify the toggle usages, we ran our tool *ts-detector* on both Chromium and Dawn source code. Following is a snippet of the JSON output of *ts-detector* resulting from command to analyze Chromium for spread toggles.

```
{
  ...
  "kUseSystemDefaultPrinter": [
    ["/Users/taj/Documents/Research/Data/chromium/
    chrome/browser/prefs/chrome_command_line_pref_store.
    cc", 1],
    ["/Users/taj/Documents/Research/Data/chromium/
    chrome/common/chrome_switches.cc", 1]
  ],
```

```
    "kNoPings": [
      ["/Users/taj/Documents/Research/Data/chromium/
     chrome/browser/prefs/chrome_command_line_pref_store.
     cc", 1],
      ["/Users/taj/Documents/Research/Data/chromium/
     chrome/common/chrome_switches.cc", 1]
    ]
  },
  "total_count": 1147
}
```

Listing 5: ts-detector output for spread toggles

The toggle variable is one of the two keys of the output JSON. Each toggle key contains a dictionary of file paths where the usage instances of that key toggle variable been has found. The total count is also collected as a separate key for convenience.

We collected these JSON outputs and stored into data tables for each four toggle usage patterns. Each table contains the file name where the toggle usage was found, and the count of how many usage instances were found. We then map the two datasets (one using *Understand*, and another using our *ts-detector*) and merge into one final dataset for our analysis.

5 RESULTS AND DISCUSSION

We analyzed the final consolidated dataset to provide answers to our research questions.

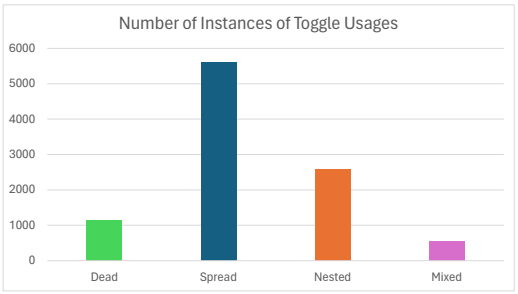

Figure 2: Number of instances of four types of Toggle Usages.

We used our tool *ts-detector* to extract four types of toggle usage instances. The bar-chart in Figure 2 shows all four types of toggle usage instances. Spread toggle is the one dominating with 5k instances because of its cross-cutting nature. Mixed usage pattern has been found the least number of times (539). Dead and nested usages are found 1.1k, and 2.5k times respectively.

Answer to RQ1—Our analysis reveal that Spread is the most frequently occurring feature toggle pattern; Mixed toggle usage pattern instances occur the least in analyzed C++ projects.

We further investigated if feature toggle usage patterns collectively exhibit any correlation with code complexity and size metrics. To understand this, we aggregated the code quality metrics in four groups: complexity metrics (CC, and MNesting) in files with toggles, and in files without toggles, code size metrics (AvgLOC, ILOC, ExLOC) in files with toggles, and in files without toggles. We compared the complexity metrics sum and code size metrics sum separately as shown in Figure **??** and Figure 3. In median case the sum of the complexity metrics (CC and MNesting) in files with toggle usages (WT) is double compared to the files without any

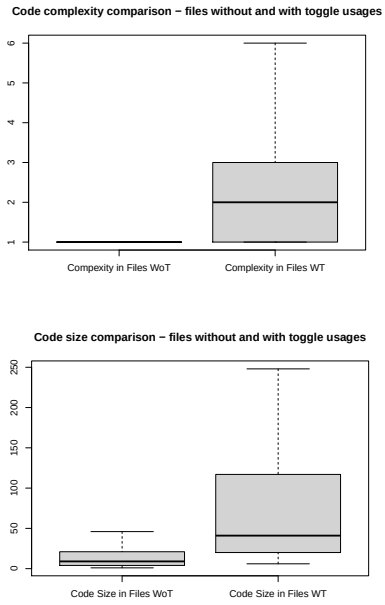

Figure 3: Comparing complexity metrics, and code size metrics between files with and without toggles usage patterns.

toggle usages (WoT). Similarly, the median code size metrics sum in files WT (10) is much higher than that (43) in files WoT.

Answer to RQ2—Feature toggle usage patterns collectively exhibit an influence on the increase of aggregated code complexity and size metrics. Files using feature toggles seem to have high number of complexity and size metrics compared to the files that do not use any feature toggles.

This led us to our last investigation (RQ3) of this paper where we wanted to understand whether there are certain toggle usage patterns that exhibit correlations with certain complexity and size metrics. We calculated Spearman correlations for each of the considered code quality metrics with each of the toggle usages. Figure 4 depicts the correlations between four toggle usage patterns (Spread, Nested, Dead, and Mixed) and the code quality metrics (CC, AvgLOC, ILOC, ExLOC, and MNesting).

The Spread usages displayed positive correlations with all of the code metrics. However, the significant correlations with ILOC, ExLOC, and MNesting are 0.56, 0.1, and 0.08 respectively with the p-values of $2.2e^{-16}$, $1.7e^{-16}$, and $7.7e^{-05}$.

The Nested usages have shown the highest correlations except for ILOC compared to the three other usage patterns. However, only two of them are statistically significant. The most significant correlations with AvgLOC, ExLOC, and MNesting are 0.5, 0.25, and 0.8 respectively, with the p-values of $7.1e^{-07}$, 0.009, and $2.2e^{-16}$.

Dead toggle usages show no correlation with any of the metrics. Similar results for Mixed toggle usages showing no significant correlation with any of the metrics.

Answer to RQ3– The Spread and Nested toggles have shown significant correlations with ILOC, ExLOC, MNesting.

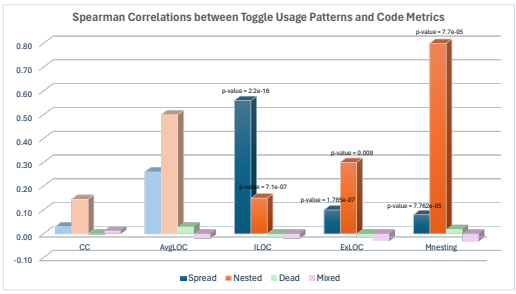

Figure 4: Correlation between toggle usage patterns and the code metrics. Insignificant correlations are faded.

It is notable that Nested toggle usages are highly correlated to "MNesting" which (highest among all) was expected because the nature of nested usage is to create nested conditions with toggle variables. The results indicates that all toggles do not have a similar influence on code quality. Since Nested and Spread usages are highly correlated to code complexity and code size metrics, we are prone to call them as toggle-smells. Further progress on this study will make us more convinced on this.

6 THREATS TO VALIDITY

Construct validity concerns with the degree to which our analyses measure what we intend to analyze. We developed a tool in this study to identify feature toggle usage patterns automatically in C++ projects. Due to the nature of feature toggles, the definition and usage of toggles are highly dependent on the development team. Hence, it is not guaranteed that one set of patterns (in the form of regular expressions) will work for all software projects. To mitigate this issue, we implement the tool using a configuration file that can be optionally provided to the tool. In the future we plan to comprehensively evaluate the tool with multiple language based projects. External validity concerns with the ability to generalize the results. Our study focuses on identifying feature toggles and their relationship with C++ code complexity. However, the analysis can be easily extended to other programming languages as our tool can look for specific patterns in provided source code based on the provided regular expressions.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we explore the frequency of feature toggle usage patterns and investigate if there is any correlation to the code-complexity. Toward this goal, we developed a tool to identify four toggle usage patterns in C++ source code. We investigated the correlation between code complexity captured via complexity metrics and the number of toggle usage patterns. Our results reveal that Nested and Spread usages are significantly correlated to the code complexity and code size. The results will motivate further results to understand the characteristics of toggles and optimize their usage. At this stage we are likely to call the Nested and Spread toggles as toggle smells, however, we need to expand this study further on all popular high-level language based source code. We also need to study the extent of nesting and spreading toggle variables before we call them toggle smells. The developed tool will help future studies to replicate and extend this study. We aim to extend our investigation to check the influence of the toggles and their density on other aspects of code quality, such as cognitive complexity, size, and testability in a future study.

REFERENCES

[1] 2022. Feature Flags. https://apptimize.com/docs/featureflags.html.
[2] Matt Campbell. 2020. Uber Open-Sources Tool to Automatically Clean Up Stale Code. https://www.infoq.com/news/2020/06/uber-piranha/. (January 2020).
[3] Mrinal Kanti Debbarma, Swapan Debbarma, Nikhil Debbarma, Kunal Chakma, and Anupam Jamatia. 2013. A review and analysis of software complexity metrics in structural testing. *International Journal of Computer and Communication Engineering* 2, 2 (2013), 129–133.
[4] Wolfram Fenske, Sandro Schulze, Daniel Meyer, and Gunter Saake. 2015. When code smells twice as much: Metric-based detection of variability-aware code smells. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 171–180.
[5] Flickr. 2009. Flipping Out. https://code.flickr.net/2009/12/02/flipping-out. (December 2009).
[6] Martin Fowler. 2017. Feature Toggles (aka Feature Flags). https://martinfowler.com/articles/feature-toggles.html. (2017).
[7] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code* (2 ed.). Addison-Wesley Professional.
[8] Dennis Kafura and Geereddy R. Reddy. 1987. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering* 3 (1987), 335–343.
[9] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. # ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 143–150.
[10] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the tenth international conference on Aspect-oriented software development*. 191–202.
[11] Kenneth Magel, Raymond Michael Kluczny, Warren A Harrison, and Arlan R Dekock. 1982. Applying software complexity metrics to program maintenance. (1982).
[12] Rezvan Mahdavi-Hezaveh, Nirav Ajmeri, and Laurie Williams. 2022. Feature toggles as code: Heuristics and metrics for structuring feature toggles. *Information and Software Technology* 145 (2022), 106813.
[13] Rezvan Mahdavi-Hezaveh, Jacob Dremann, and Laurie Williams. 2021. Software development with feature toggles: practices used by practitioners. *Empirical Software Engineering* 26, 1 (2021), 1–33.
[14] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2017. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering* 44, 5 (2017), 453–469.
[15] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. 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*. 233–242.
[16] Netflix. 2018. Preparing the Netflix API for Deployment. https://netflixtechblog.com/preparing-the-netflix-api-for-deployment-786d8f58090d. (November 2018).
[17] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C Rigby, and Bram Adams. 2016. Feature toggles: practitioner practices and a case study. In *Proceedings of the 13th international conference on mining software repositories*. 201–211.
[18] Tajmilur Rahman. 2023. Feature Toggle Usage Patterns: A Case Study on Google Chromium. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 142–147.
[19] Scitools. [n. d.]. Understand's Python API. https://documentation.scitools.com/html/python/index.html.
[20] Scitools. 2024. Understand. https://scitools.com/.
[21] Doug Seven. 2014. Knightmare: A DevOps Cautionary Tale. https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/. (April 2014).
[22] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173. https://doi.org/10.1016/j.jss.2017.12.034
[23] sillysaurus3. 2017. Feature flags are a nice idea, but nobody outside of Facebook seems to be embracing them. https://news.ycombinator.com/item?id=15377207. (October 2017).
[24] Thomas Steiner. 2021. How to set browser flags in Chromium. https://developer.chrome.com/blog/browser-flags/. (May 2021).
[25] Xhevahire Tërnava, Luc Lesoil, Georges Aaron Randrianaina, Djamel Khelladi, and Mathieu Acher. 2022. On the Interaction of Feature Toggles. In *VaMoS 2022-16th International Working Conference on Variability Modelling of Software-Intensive Systems*.
[26] Tong Yi and Chun Fang. 2020. A complexity metric for object-oriented software. *International Journal of Computers and Applications* 42, 6 (2020), 544–549.