ASSESSING THE EVOLUTION OF COMPLEX METHODS: A MULTI-LANGUAGE STUDY

MATEUS FELLIPE ALVES LOPES

ASSESSING THE EVOLUTION OF COMPLEX METHODS: A MULTI-LANGUAGE STUDY

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Andre Cavalcante Hora

Belo Horizonte

Agosto de 2021

MATEUS FELLIPE ALVES LOPES

ASSESSING THE EVOLUTION OF COMPLEX METHODS: A MULTI-LANGUAGE STUDY

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Andre Cavalcante Hora

Belo Horizonte
August 2021

© 2021, Mateus Fellipe Alves Lopes.
Todos os direitos reservados

Lopes, Mateus Fellipe Alves.

L864a Assessing the evolution of complex methods: a multi-l language study [manuscrito] / Mateus Fellipe Alves Lopes. – 2021.
xiv, 76 f. il.

Orientador: Andre Cavalcante Hora.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.69-76
.
1. Computação – Teses. 2. Software – Codificação –Teses. 3. Software – Manuteção – Teses. 4. Software – Desenvolvimento – Teses. I. Hora, Andre Cavalcante. II.Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Assessing the Evolution of Complex Methods: A Multi-Language Study

MATEUS FELLIPE ALVES LOPES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ANDRÉ CAVALCANTE HORA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 3 de Setembro de 2021.

Acknowledgments

Agradeço a todos que me ajudaram e contribuíram com minha jornada ao longo deste mestrado. Este trabalho não teria sido possível sem o seu apoio, incentivo e orientação. Em especial, gostaria de agradecer:

A Deus, por estar sempre guiando e protegendo a minha caminhada, tornando possível esse momento vivido.

Aos meus pais, Benjamin e Cleonice, e a minha irmã, Raquel, que sempre ofereceram apoio e motivação para que eu não me deixasse abater pelas dificuldades e continuar em busca de meus objetivos.

Ao meu orientador Andre Hora, que contribuiu de forma significativa para a realização desse trabalho, me guiando e fornecendo todo suporte necessário para meu crescimento pessoal e profissional.

Aos meus colegas do ASERG, pelo acolhimento e pelas trocas de conhecimento durante o decorrer deste período.

Aos meus amigos, pela amizade, apoio e momentos felizes proporcionados.

Aos membros da banca, Prof. Eduardo Figueiredo e Prof. Marco Tulio Valente, pela disponibilidade em participar deste trabalho.

Ao DCC/UFMG e a CAPES, pelo suporte financeiro, logístico e profissional.

""Try to move the world — the first step will be to move yourself.""

(Plato)

Resumo

Ao longo do tempo, os sistemas de software tendem a aumentar a complexidade e se tornarem mais difíceis de manter. Apesar das desvantagens da complexidade em código serem bem conhecidas, código complexo está presente na maioria dos projetos de software reais. Portanto, surge uma questão importante: por que, com todos os conselhos que existem contra essa prática, métodos complexos continuam a existir? Infelizmente, a complexidade de código é normalmente avaliada para uma única linguagem de programação (geralmente Java), reduzindo a generalidade das descobertas. Assim, avaliar como e por qual razão códigos complexos evoluem em múltiplas linguagens de programação é fundamental para entender as semelhanças e diferenças entre as linguagens. Nesta dissertação, fornecemos um estudo empírico multilinguagem para avaliar a evolução de métodos complexos e um estudo para entender melhor as percepções dos desenvolvedores. Analisamos 1.000 métodos complexos de 50 projetos populares escritos em JavaScript, Python, Java, C++ e C# e realizamos uma pesquisa com mais de 70 desenvolvedores, incluindo desenvolvedores de grandes empresas, como Google, Facebook e Apple. Descobrimos que a linguagem de programação desempenha um papel importante no estudo da complexidade de código e que os métodos complexos não são homogêneos nas operações que realizam. A percepção de complexidade dos desenvolvedores é subjetiva e varia de acordo com a linguagem de programação. Além disso, os desenvolvedores podem evitar deliberadamente a refatoração de código complexo devido a vários motivos, incluindo estabilidade do código, falta de prioridade e risco de refatoração. Finalmente, com base em nossas descobertas, discutimos ideias e aplicações para pesquisadores e profissionais.

Palavras-chave: Complexidade de Código, Code Smells, Manutenção de Software, Evolução de Software, Estudo Empírico.

Abstract

Over time, software systems tend to increase complexity and become harder to maintain. Despite the drawbacks of code complexity are well-known, complex code is present in most real software projects. Here, an important question arises: why, with all the advice out there against it, do we continue to end up with complex methods? Unfortunately, code complexity is typically assessed for single programming languages (often Java), reducing the generality of findings. Thus, assessing how and why complex code evolves in multiple programming languages is fundamental to better understand the similarities and differences among the languages. In this dissertation, we provide a multi-language empirical study to assess the evolution of complex methods and a survey study to better understand developers' perceptions. We analyze 1,000 complex methods of 50 popular projects written in JavaScript, Python, Java, C++, and C# and we perform a survey with over 70 developers, including developers from large companies, like Google, Facebook, and Apple. We find that programming language plays an important role in the study of code complexity and that complex methods are not homogeneous in the operations they perform. The developers' perception of complexity is subjective and varies per programming language. Moreover, developers may deliberately avoid refactoring complex code due to several reasons, including code stability, lack of priority, and refactoring risk. We conclude by discussing insights for researchers and practitioners.

Palavras-chave: Code Complexity, Code Smells, Software Maintenance, Software Evolution, Empirical Study.

List of Figures

2.1 Example method from the CPython repository, written in Python. 7
2.2 Example method from the Dubbo repository, written in Java. 8
2.3 Data is more precise when a single entity is the target. Top: complexity at system level over time. Bottom: complexity at method level over time. . . 10
2.4 Data is less likely to be impacted by external noise. Left: complexity at system level over time. Right: complexity at method level over time. . . . 11

3.1 Distribution of the complexity of the selected methods. 21
3.2 Distribution of the complexity per NLOC of the selected methods. 22
3.3 Distribution of commits of the selected methods. 23
3.4 Word cloud of the developers' responses. 26

4.1 Distribution of complexity in the first, intermediate, and last versions. . . . 33
4.2 Trend analysis. (left): proportion trend categories. (right): distribution of commits per trend category 34
4.3 Examples of complex methods per category. (left): increasing complexity. (center): decreasing complexity. (right): same complexity. 34
4.4 Distribution of project size and code size. 35
4.5 Collection method example (project ILSpy). 43
4.6 Conversion method example (project Moment). 44
4.7 Coordination method example (project ElasticSearch). 45
4.8 Accessing method example (project Nancy). 46

5.1 Developers who consider the methods as complex. 51
5.2 Developers who do not consider the methods as complex. 52
5.3 Proportion of answers per programming language. 53
5.4 Developers' insights about how hard is to change complex methods. 54
5.5 Reasons why complex methods are not refactored. 56

6.1 Experience of the surveyed developers vs. perceptions of complexity. 61

6.2 Complexity of the analyzed methods. 62
6.3 Distribution of (a) messages and (b) changed files in issues/PRs. 64

List of Tables

2.1 Summary of studies by target language. 15

3.1 Selected software systems (Size: number of source files). 18
3.2 Correlation between complexity and NLOC. 22
3.3 Most complex methods per language. 23
3.4 Overview of the respondent developers. 26

4.1 Code changes in complex methods. 36
4.2 Code changes in complex methods per language (%). 36
4.3 Change classification in complex and not complex methods. 37
4.4 Change classification in complex methods by language. 37
4.5 Residuals for bugs. 38
4.6 Residuals for new features. 38
4.7 Residuals for refactoring. 38
4.8 Content categories of the complex methods. 39
4.9 Concentration of complex methods per content category. Concentration column: Low: ratio < 1; Medium: $1 \leq$ ratio < 2; High: ratio ≥ 2. *Frequency of "All Methods" is based on [27]. 40
4.10 Categories of the complex methods by language. 40
4.11 Growth of complex methods per category. Growth column: Low: ratio $<$ 1.2; Medium: $1.2 \leq$ ratio < 1.4; High: $1.4 \geq$ ratio. Commits, first, and last are median values. 41

6.1 Code and evolution of the complex methods (median values). 63
6.2 Maintenance problems found in issues and pull requests of complex methods. 65

Contents

Acknowledgments **vi**

Resumo **viii**

Abstract **ix**

List of Figures **x**

List of Tables **xii**

1 Introduction **1**
1.1 Motivation and Problem 1
1.2 Proposed Work 2
1.3 Contributions 3
1.4 Outline of the Dissertation 4

2 Background and Related Work **6**
2.1 Cyclomatic Complexity 6
2.2 Complexity at Method/Function Level 9
2.2.1 Benefits 9
2.2.2 Challenges 10
2.3 Code Smells and Complexity 11
2.4 Related Work 13
2.5 Final Remarks 16

3 Study Design **17**
3.1 Selecting Software Systems 17
3.2 Extracting Methods and Computing Complexity 19
3.3 Exploring the Complex Methods 21
3.4 Assessing Evolution of Complex Methods 23

3.5 Survey Analysis . . . 24
3.5.1 Data Collection . . . 24
3.5.2 Data Analysis . . . 26
3.6 Research Questions . . . 27
3.7 Final Remarks . . . 31

4 Empirical Results: Evolution of Complex Methods 32
4.1 RQ1: How do complex methods evolve over time? . . . 32
4.2 RQ2: What changes are performed on complex methods? . . . 37
4.3 RQ3: What operations are implemented in complex methods? Which ones are more likely to become more complex? . . . 39
4.4 Discussion e Implications . . . 47
4.5 Threats To Validity . . . 47
4.6 Final Remarks . . . 49

5 Survey Results: Developers' Perceptions on Complex Methods 50
5.1 RQ4: What are developers' perceptions of method complexity? . . . 50
5.2 RQ5: Why complex methods are not eliminated from code? . . . 56
5.3 Discussion and Implications . . . 58
5.4 Threats To Validity . . . 59
5.5 Final Remarks . . . 59

6 Assessing Self-Admitted Complex Methods 60
6.1 RQ6: To what extent are self-admitted complex methods different from other complex methods? . . . 60
6.1.1 Developer Experience . . . 61
6.1.2 Code and Evolution . . . 62
6.1.3 Maintenance . . . 63
6.2 Discussion e Implications . . . 65
6.3 Threats To Validity . . . 66
6.4 Final Remarks . . . 66

7 Conclusion 67
7.1 Summary and Contributions . . . 67
7.2 Future Work . . . 68

Bibliography 69

Chapter 1

Introduction

1.1 Motivation and Problem

During software evolution, new features are added, bugs are fixed, and code is adapted due to the changing environment. Often, those changes are performed without the proper care, that is, developers do not apply refactoring nor code cleaning, decreasing software quality [22; 46]. Consequently, over time, software systems tend to increase their complexity and become harder to maintain [38; 39; 40]. Indeed, developers recognize that code smells related to complexity are the most harmful for maintenance [70]. Despite the drawbacks of code complexity are well-known, complex code is present in most real software projects. Here, an important question arises: *why, with all the advice out there against it [22; 46], do we continue to end up with complex methods in software systems?*

Better understanding the evolution of complex code (and other smells [22]) is fundamental to detect risk areas within a software system that need attention and to drive future development activities [8; 9; 20; 22; 46; 51; 52; 53; 72; 73; 78]. Typically, studies on code complexity (and code smells) are restricted to a single programming language, mostly Java (*e.g.,* [9; 15; 20; 34; 41; 49; 51; 52; 53; 54; 56; 65; 72; 73; 74; 77; 79]). Unfortunately, this Java-focus causes a lack of empirical analysis on other popular programming languages, reducing the generality of their findings. For example, languages like JavaScript and Python remain under-studied, and we are unsure whether Java observations apply to those languages. Furthermore, distinct languages have different programming styles, for example, JavaScript and Python programs are not necessarily written with the OO paradigm.

In addition to the analysis of a single programming language, few studies assess complexity at a fine level of granularity (*e.g.,* [30; 41]), for example, observing methods

instead of classes. Assessing data at this level of granularity has some benefits and challenges. For example, analyzing a fine-grained entity, such as a method, can provide fine and precise information on how *that* entity is affected. In contrast, system level analysis can only provide an overall view. Another benefit we notice when assessing fine-grained entities is that they are less affected by external changes. In contrast, at the system level, code changes not necessarily related may bias the analysis [8; 9; 51]. Thus, there is a trade off between coarse and fine-grained analysis. The former produces less data as the whole system or few large components can be assessed individually. The latter generates more information to be analyzed as thousands of methods may exist even in a small software project.

Thus, those aspects may impact the way developers perceive and handle code complexity. Assessing complex code at the method level and its evolution in multiple programming languages is fundamental (1) to increase the generality of findings, (2) to better understand the similarities and differences among the languages, and (3) to reason why complexity remains in code regardless of well-known best coding practices.

1.2 Proposed Work

In this dissertation, we conduct a multi-language *empirical study* to assess the evolution of complex methods and a *survey study* to better understand developers' perceptions on complex methods. First, we analyze 1,000 complex methods provided by 50 popular projects that are written in five programming languages (JavaScript, Python, Java, C++, and C#) and analyze how those methods achieved such a state. We propose research questions to assess their evolution, change type, and operation, as follows:

- *RQ1: How do complex methods evolve over time?* Complex methods tend to become even more complex over time, while the effort to decrease it is minimum. However, this varies per programming language: C++ and Python projects have more methods that increase complexity, whereas Java and C# present more efforts to reduce it.

- *RQ2: What changes are performed on complex methods?* Complex methods are significantly over-represented on changes related to bugs, new features, and refactoring. For example, complex methods undergo 2.2x more refactoring than non-complex ones.

- *RQ3: What operations are implemented in complex methods and which ones are more likely to become more complex?* Complex methods are over-represented on

certain operations, namely *processing*, *conversion*, *validation*, and *IO*. However, the growth in complexity largely varies according to the operations.

Overall, the results show that, independently of the programming language, complex methods tend to become more complex over time. Thus, to explore the reasons behind it, we perform a survey with over 70 developers who have maintained complex methods in JavaScript, Python, Java, C++, and C#, including developers from Google, Facebook, Apple, Microsoft, and Oracle. Specifically, we seek to understand the impact of complexity on their maintenance tasks and the reasons for the existence of complex methods. Thus, we propose the following research questions:

- *RQ4: What are developers' perceptions of method complexity?* Half of the surveyed developers do not consider complex methods (as measured by Cyclomatic Complexity) as complex, while 40% of the participants recognize the problems associated with method complexity.

- *RQ5: Why complex methods are not eliminated from code?* Developers provide several reasons on why complex methods remain in code, including *code is stable*, *refactoring is not a priority*, *refactoring is risky*, *problem is inherently complex*, *code is rarely changed*, and *code flow must be shown.*

Lastly, based on the survey data, we perform a complementary study to explore then notion of *self-admitted complex methods*, that is, methods that the developers themselves classify as complex. Therefore, we propose the following research question:

- *RQ6: To what extent are self-admitted complex methods different from other complex methods?* We find that the experience of the developers may play a role in their perceptions of code complexity. However, we could not find any major difference between self-admitted complex methods and other complex methods in their code, evolution, and maintenance.

1.3 Contributions

The contributions of this research are fourfold:

- We provide a large multi-language empirical study to assess the evolution of complex methods.

- We provide a survey with developers from worldwide companies to assess their perceptions on the methods considered complex.
- We provide a preliminary empirical study to assess the characteristics of self-admitted complex methods.
- We provide a set of findings and insights to researchers and practitioners.

Our results provide insights to both researchers and practitioners. We present that the studied programming languages have close but not equal issues regarding method complexity, thus, researchers should not focus their analysis on single languages. We present that, independently of the programming language, complex methods are living entities that change frequently and grow in complexity over time. Our survey results show that the perception of complexity is somehow subjective and varies per programming language. For example, C++ and Python have the highest rate of methods increasing complexity, while their developers have the lowest perception of complexity. We also provide and discuss 10 reasons on why complex methods are deliberately not refactored over time. Finally, we provide results towards novel studies to automatically identify self-admitted complex methods.

1.4 Outline of the Dissertation

The remaining of this dissertation is organized as follows:

- **Chapter 2** provides an overview of the main concepts related to this dissertation, covering topics as code complexity and code smells. We also discuss the related work, comparing the major differences between the literature and our study.
- **Chapter 3** describes the methodology. First, we present how we select the studied software systems in multiple programming languages. Next, we describe the method used to extract and monitor the target methods. Then, we show the procedure used to create, apply, and analyze the survey. Finally, we present the techniques and strategies used to evaluate self-admitted complex methods.
- **Chapter 4** presents the results of the empirical study (*RQ1*, *RQ2*, and *RQ3*), which involves the evolution of the complexity of the methods. For each question, we summarize the major findings. We conclude by describing practical implications and threats to validity.

- **Chapter 5** describes the survey results, answering *RQ4* and *RQ5*. Through the survey responses, it is possible to identify the real perception of the developers of methods considered complex by the literature. At the end of the chapter, we list implications and threats to validity of the survey.

- **Chapter 6** presents the results of the final research question (*i.e., RQ6*). The analysis presents some comparisons between the self-admitted complex and not complex methods. We conclude by discussing implications and threats to validity.

- **Chapter 7** presents the conclusion of this dissertation, including an overview of the study and future work.

Chapter 2

Background and Related Work

In this chapter, we provide an overview to understand the study presented in this dissertation. First, Section 2.1 discusses and exemplifies the Cyclomatic Complexity metric. Section 2.2 presents the benefits and challenges of addressing complexity at the method level. Next, we introduce a related concept, code smells, in Section 2.3. Then, Section 2.4 presents the related work. Finally, we conclude the chapter in Section 2.5.

2.1 Cyclomatic Complexity

One of the most popular metrics involving source code complexity is Cyclomatic Complexity, developed by Thomas J. McCabe in 1976 [48]. The author's objective was to compute the complexity of a source code statically, analyzing the program's control flow. The value obtained through the Cyclomatic Complexity is equivalent to the maximum number of linearly independent paths of the code. In this way, the higher the value of the metric, the more possible paths there are. Consequently, increasing the effort needed to read, understand, and test the functionality of a method, class, or module in the software system [22; 37; 46].

Thus, the Cyclomatic Complexity metric can be calculated simply by the Formula 2.1, where *number_decisions* indicates the count of conditional statements, such as commands: `if`, `else if`, `case`, `for`, `while`, and boolean operators (`&&` and `||`). The last one was not part of the original metric but added in a new calculus extension.

$$cyclomatic_complexity = number_decisions + 1 \tag{2.1}$$

To illustrate the computation of Cyclomatic Complexity in source code, we selected two methods of real software systems that use distinct components that impact

their complexity. The first example, Figure 2.1, presents a method that identifies certain types of tags in HTML and assigns state values according to the tags. This code is written in Python and belongs to the CPython repository. The method consists of 33 conditional statements (*i.e.*, 10 `if`, 14 `elif`, 8 boolean operators, and one `for`), resulting in a Cyclomatic Complexity of 34. Note that the flow caused by the `else` command is computed by the sum of one unit of the Formula 2.1, not being calculated individually like the other conditionals.

```
def handle_starttag(self, tag, attrs):
    "Handle starttags in help.html."
    class_ = ''
    for a, v in attrs:
        if a == 'class':
            class_ = v
    s = ''
    if tag == 'div' and class_ == 'section':
        self.show = True    # Start main content.
    elif tag == 'div' and class_ == 'sphinxsidebar':
        self.show = False   # End main content.
    elif tag == 'p' and self.prevtag and not self.prevtag[0]:
        # Begin a new block for <p> tags after a closed tag.
        # Avoid extra lines, e.g. after <pre> tags.
        lastline = self.text.get('end-1c linestart', 'end-1c')
        s = '\n\n' if lastline and not lastline.isspace() else '\n'
    elif tag == 'span' and class_ == 'pre':
        self.chartags = 'pre'
    elif tag == 'span' and class_ == 'versionmodified':
        self.chartags = 'em'
    elif tag == 'em':
        self.chartags = 'em'
    elif tag in ['ul', 'ol']:
        if class_.find('simple') != -1:
            s = '\n'
            self.simplelist = True
        else:
            self.simplelist = False
        self.indent()
    elif tag == 'dl':
        if self.level > 0:
            self.nested_dl = True
    elif tag == 'li':
        s = '\n* ' if self.simplelist else '\n\n* '
    elif tag == 'dt':
        s = '\n\n' if not self.nested_dl else '\n'  # Avoid extra line.
        self.nested_dl = False
    elif tag == 'dd':
        self.indent()
        s = '\n'
    elif tag == 'pre':
        self.pre = True
        if self.show:
            self.text.insert('end', '\n\n')
        self.tags = 'preblock'
    elif tag == 'a' and class_ == 'headerlink':
        self.hdrlink = True
    elif tag == 'h1':
        self.tags = tag
    elif tag in ['h2', 'h3']:
        if self.show:
            self.header = ''
            self.text.insert('end', '\n\n')
        self.tags = tag
    if self.show:
        self.text.insert('end', s, (self.tags, self.chartags))
    self.prevtag = (True, tag)
```

Figure 2.1. Example method from the CPython repository, written in Python.

Figure 2.2 presents another example of a method with several elements that increase its complexity. This one is written in Java, from the Dubbo repository. This method has a Cyclomatic Complexity of 23, having the functionality to manipulate method parameters sent by an object in its signature. In addition, the source code contains several nested conditionals, such as a `try-catch` that is also taken into account when calculating the complexity of the method.

```java
public static void appendParameters(Map<String, String> parameters, Object config, String prefix) {
        if (config == null) {
            return;
        }
        Method[] methods = config.getClass().getMethods();
        for (Method method : methods) {
            try {
                String name = method.getName();
                if (MethodUtils.isGetter(method)) {
                    Parameter parameter = method.getAnnotation(Parameter.class);
                    if (method.getReturnType() == Object.class || parameter != null && parameter.excluded()) {
                        continue;
                    }
                    String key;
                    if (parameter != null && parameter.key().length() > 0) {
                        key = parameter.key();
                    } else {
                        key = calculatePropertyFromGetter(name);
                    }
                    Object value = method.invoke(config);
                    String str = String.valueOf(value).trim();
                    if (value != null && str.length() > 0) {
                        if (parameter != null && parameter.escaped()) {
                            str = URL.encode(str);
                        }
                        if (parameter != null && parameter.append()) {
                            String pre = parameters.get(key);
                            if (pre != null && pre.length() > 0) {
                                str = pre + "," + str;
                            }
                        }
                        if (prefix != null && prefix.length() > 0) {
                            key = prefix + "." + key;
                        }
                        parameters.put(key, str);
                    } else if (parameter != null && parameter.required()) {
                        throw new IllegalStateException(config.getClass().getSimpleName() + "." + key + " == null");
                    }
                } else if (isParametersGetter(method)) {
                    Map<String, String> map = (Map<String, String>) method.invoke(config, new Object[0]);
                    parameters.putAll(convert(map, prefix));
                }
            } catch (Exception e) {
                throw new IllegalStateException(e.getMessage(), e);
            }
        }
    }
```

Figure 2.2. Example method from the Dubbo repository, written in Java.

The evaluation of Cyclomatic Complexity metric is subjective and may vary according to context, just like any other quality metric in software engineering. However,

based on the analysis and examples of McCabe and Watson [47], the industry adopts some limits to be observed and monitored to detect possible quality problems [6]. Thus, the most accepted complexity thresholds are:

- 1 - 10: Simple method. **Low risk**;
- 11 - 20: Reasonably complex method. **Moderate risk**;
- 21 - 50: Very complex method. **High risk**;
- 51+: Very unstable method. **Very high risk**.

Indeed, mitigation of the excessive presence of conditionals in source code is so important that Fowler has devoted an entire chapter of his book to simplifying conditional expressions by performing refactoring operations [22] (*e.g., Decompose Conditional*, *Replace Conditional with Polymorphism*, and *Consolidate Duplicate Conditional Fragments*).

2.2 Complexity at Method/Function Level

Overall, complexity can be assessed at coarse-grained levels (*e.g.,* system and class) or fine-grained levels (*e.g.,* method and function). Mining code complexity at the method/function level provides some benefits and challenges. Next, we briefly discuss those benefits in the light of concrete examples.

2.2.1 Benefits

Data is more precise when a single entity is a target. Assessing a fine-grained entity, such as a method, can provide fine and precise information on how *that* entity is affected. In contrast, system level analysis can only provide an overall view. For instance, consider the examples presented in Figure 2.3, which shows the metric Cyclomatic Complexity at both system and method level over time (systems are presented on the top charts and methods on the bottom ones). Notice that the overall system complexity is increasing for the three projects presented on the top. Even in relative terms, these trends would not provide detailed insights. However, at the method level, the trends can be completely distinct, increasing, constant, or even decreasing complexity over time (see the three charts on the bottom). That is, at the system level, fine-grained changes that happen at the method level may not be noticeable [25].

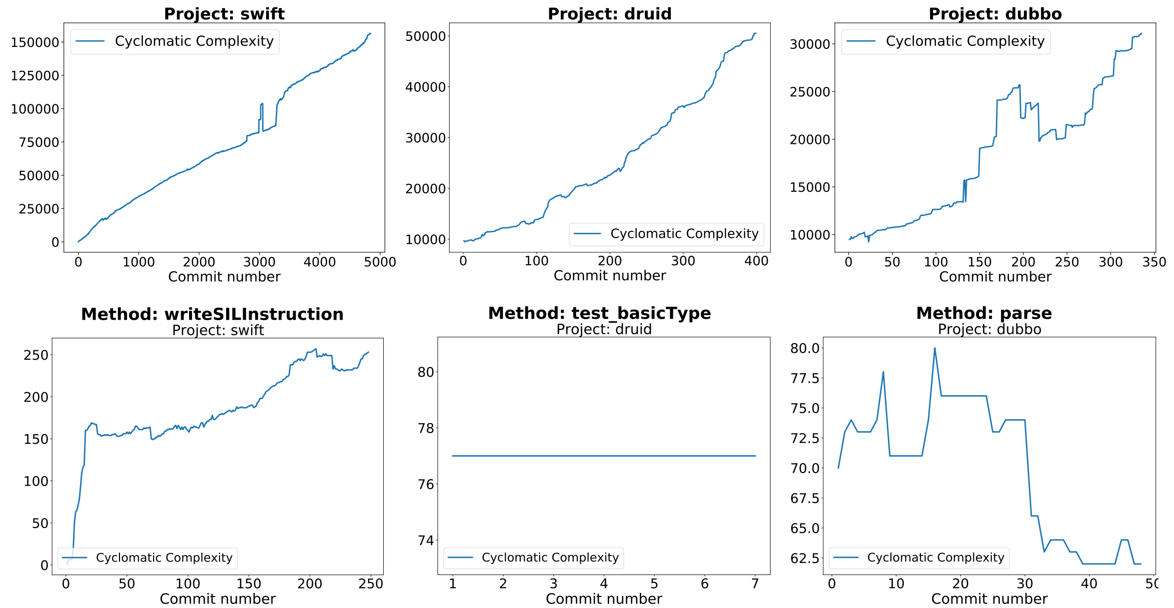


Figure 2.3. Data is more precise when a single entity is the target. Top: complexity at system level over time. Bottom: complexity at method level over time.

Data is less likely to be impacted by external noise. Another benefit we notice when assessing fine-grained entities is that they are less affected by external changes. In contrast, at the system level, code changes not necessarily related may bias the analysis [8; 9; 51]. Consider the examples presented in Figure 2.4: on the left side, we observe that the sudden drop in the overall system complexity happens simply because methods were removed. That is, the system complexity did not decay due to refactoring nor code cleaning operations. On the other hand, at the method level (right side), the drop in complexity is more likely to indicate a real refactoring or code cleaning since the method itself remains while its code is changed. That is, at the method level, a single entity and its content are tracked, thus, any change in its content is a real one [28]. In contrast, at the system level, multiple entities and multiple contents are tracked, thus, changes are harder to control, causing noise to the analysis.

2.2.2 Challenges

Data is more frequent. Clearly, there is a trade off between coarse and fine-grained analysis. The former produces less data as the whole system or few large components can be assessed individually. The latter generates more information to be analyzed as thousands of methods may exist even in a small software project. Thus, to overcome this issue, we may need to focus on analyzing some entities instead of all entities.

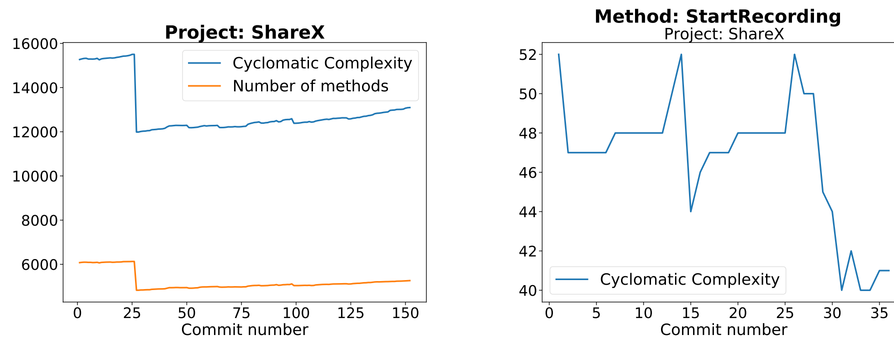


Figure 2.4. Data is less likely to be impacted by external noise. Left: complexity at system level over time. Right: complexity at method level over time.

Data is more likely to be impacted by internal noise. Another challenge we may face when assessing fine-grained entities is the impact caused by internal changes, such as entity renaming and lack of version history. For example, suppose we are analyzing the complexity of a method over time, and, at some point, this method is renamed. As expected, this can disturb the analysis: if the rename operation is not resolved, it may be misdetected as a method removal, and its history is missed [28]. For instance, suppose a complex method `foo()` with a long history that was recently renamed to `bar()`; in this case, one may misdetect that `bar()` is a brand new method that was created as complex, and not as a method that became complex over time. Indeed, detecting operations, such as renaming, is not a trivial task. Some refactoring detection tools were recently proposed with good precision, however, they mostly target Java systems [61; 62; 63; 71].

Due to the aforementioned benefits (*i.e.*, data is more precise and frequent and data less prone to be impacted by external noise), in this dissertation, we assess code complexity at method/function level rather than class or system level. However, we also pay special attention to handle the mentioned challenges.

2.3 Code Smells and Complexity

In Software Engineering, a code smell is any feature in the source code that possibly indicates a deeper problem [22]. The name is intended to allude to a bad smell from the real world, which bothers and indicates something damaged or problematic. The term code smell (or bad smell) was introduced by Kent Beck and has become popular due to Fowler's book on refactoring [22]. In addition to listing 22 code smells, the author listed the operations necessary to remove and mitigate each code smell. Fowler also

defined code smell as "*A code smell is a surface indication that usually corresponds to a deeper problem in the system*".[1]

Code smells can be classified according to their granularity to system components: (1) application (*e.g., Shotgun Surgery* and *Duplicated Code*), (2) classes (*e.g., Large/God Class*, *Lazy Class*, and *Data Class*), and (3) methods (*e.g., Long Parameter List*, *Long Method*, and *Switch Statements*). Thus, developers can focus their efforts on certain contexts according to the needs (code smell names may vary by author, but the principles are the same). In general, the identification of a code smell is subjective and change according to each context and experience of Software Engineers. The main method for removing a code smell is refactoring [22].

We observe that some code smell definitions are closely related to code complexity [22; 37], either by the direct calculation of the *cyclomatic complexity* for the method/function or via the *weighted method count* metric (which evaluates the complexity at the class level, considering the sum of the complexity of all the methods that belong to that class [12]). Next, we briefly describe those code smells:

- *Brain Method*: methods that tend to centralize the functionality of a class with excessive branching, many variables used, and many lines of code. Thus, the method became difficult to understand, maintain, and increasingly complicated to evolve. We can identify theses methods by combining the metrics: (1) *Lines of Code*; (2) *Cyclomatic Complexity*; (3) *Number of Accessed Variables. Extract Method*, *Introduce Parameter Object*, and *Decompose Conditional* are possible refactoring operations that can mitigate this code smell.

- *God Class*: when a class is trying to do too much, they are large and complex. This harms the reusability and the understandability of that part of the system. The metrics commonly used for its detection are: (1) *Access To Foreign Data*; (2) *Tight Class Cohesion*; (3) *Weighted Method Count.* The two main refactoring operations for removing this code smell are *Extract Class* and *Extract Subclass.*

- *Data Class*: classes that have fields, getting and setting methods for the fields, and nothing else. They are "dumb" data holders without complex functionality. These classes break the encapsulation and data hiding principle of object-oriented programming, allowing other classes to manipulate their data. The metrics used to identify this code smell are: (1) *Number of Accessor Methods*; (2) *Number of Public Attributes*; (3) *Weighted Method Count*; (4) *Weight of a Class.* Some

[1] Fowler's blog: `https://martinfowler.com/bliki/CodeSmell.html`

ways to reduce possible impacts of this code smell are via *Encapsulate Field*, *Encapsulate Collection*, and *Remove Setting Method*.

- *Refused Parent Bequest*: when a child class refuses to use special bequest prepared by its parent, becoming a large and complex child class. This code smell interferes with one of the inheritance principles. The main metrics for identifying this anti-pattern are: (1) *Average Method Weight*; (2) *Base Class Overriding Ratio*; (3) *Base Class Usage Ratio*; (4) *Number of Methods*; (5) *Number of Protected Members*; (6) *Weighted Method Count*. *Push Down Method* and *Push Down Field* are the two main refactorings used to correct class hierarchies.

In this dissertation, we focus on *cyclomatic complexity* as we work at the level of methods and functions.

2.4 Related Work

Many studies have investigated the impact and evolution of code smells. Although code complexity is not directly a code smell, it may be related to several ones, such as long method, large class, duplicated code, etc. Tufano *et al.* [72] reported the results of a large-scale empirical study conducted on the evolution history of 200 open source projects of three software ecosystems written in Java. Five code smells are investigated at the class level (*i.e.,* Blob Class, Class Data Should be Private, Complex Class, Functional Decomposition, and Spaghetti Code). Their findings indicate a contradiction to common sense, stating that smells are introduced when the artifacts are created. In a follow-up study [73], the authors detected that 80% of the smells continue in the system over time. Olbrich *et al.* [51] investigated the evolution of code smells and their impact on the changed behavior, at the class level. Two code smells, God Class and Shotgun Surgery, are selected by analyzing two projects written in Java. The results showed that it is possible to identify different phases in the evolution of code smells. Palomba *et al.* [54] proposed an approach to detect five code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy, by exploring class change history in eight Java systems.

Other studies have been proposed to identify code smells using machine learning [13; 15; 21; 34; 44], but Di Nucci [15] alerts that machine learning techniques have critical limitations, which deserve further research.

Liu *et al.* [41] evaluated a preliminary study of how Cyclomatic Complexity changes during successive software versions. The authors selected six open source

projects written in Java and analyzed the complexity at three levels (*i.e.,* method, class, and system level). The results showed that the complexity observed across classes and methods varied less than the complexity assessed by the system as a whole. Meanwhile, Jbara *et al.* [30] identified functions with a high Cyclomatic Complexity in the Linux Kernel, written in C. The results showed that a high Cyclomatic Complexity is not considered a problem in most functions in the Linux system. They also noticed that the developers do not consider complex structures with long switch cases and successive if statements.

Yamashita and Moonen [79] investigated interactions among twelve different code smells. To this aim, professional developers were hired to implement requests on four Java systems with known smells. The authors found that the interaction occurs between code smells distributed by "coupled smells". Oizumi *et al.* [50] seek to relate code smells agglomeration with system design problems. The authors analyzed more than 2,200 agglomerations in seven projects written in Java. In this way, they confirmed that code anomalies often "flock together" to embody a design problem. Macia *et al.* [42] also identified symptoms of architecture degradation directly related to code smells. The work identified that more than 70% of the architectural problems were related to code smells. They evaluated six projects written in Java, C++, and C#.

Khomh *et al.* [33] presented a study analyzing the relations between 29 code smells and changes occurring to classes written in Java and C++ at the release level. The authors detected that code smells have a negative impact on classes. Olbrich *et al.* [52] evaluated three open source systems written in Java and find a similar behavior on God and Brain Classes. However, when they normalized with respect to size, then God and Brain Classes were less subject to change and defects. Sobrinho *et al.* [67] conducted an extensive review of the literature on code smells. They found that some code smells are more studied than others, in particular Duplicated Code, Large Class, Feature Envy, and Long Method. Fernandes *et al.* [18] also carried out a review of the literature on code smells, searching for the main tools that detect code smells. Thus, they found 84 tools, of which most tools were written and analyze Java source codes.

Some studies focus on other programming languages than Java. Researchers have analyzed distinct code smells in projects written in JavaScript [17; 31; 60]. For example, Saboury *et al.* [60] and Johannes *et al.* [31] evaluated the negative presence of bad smells in JavaScript files. In the Python language, Vavrová and Zaytsev [75] and Chen *et al.* [10] investigated code smells and anti-patterns. The result showed that the number of code smells tends to increase with several versions of the Python system. Finally, Lanza [36] proposed an evolution matrix to visualize the class history written in Smalltalk.

Other studies looked at the developers' perception of code smells. Yamashita and Moonen [78] conducted exploratory research to investigate the knowledge of code smell; the result shows that a great number of professionals do not care about them in a survey involving 85 professional software developers. Similarly, Peters and Zaidman [56] and Taibi *et al.* [70] demonstrate that developers do not treat code smells with high priority. Meanwhile, Palomba *et al.* [53] demonstrated that the developer experience and knowledge about the system are important factors for identifying smells, by analyzing answers from Master's students and professional developers.

Another topic related to the perception of developers and code smells is the *self-admitted technical debts*, which have recently been extensively studied. Potdar and Shihab [57] introduced this term by observing developers' behavior in documenting poor solutions through source code comments. In this way, they found that 3%-31% of source code files contain self-admitted technical debt, and 26%-31% of them get removed. Maldonado *et al.* [43] proposed a technique to identify self-admitted technical debt, based on fixed keywords and phrases. The proposed technique achieved accuracy for between 80%-90% of the cases. Recently, Xavier *et al.* [76] identified self-admitted technical debt in GitHub issues, finding they take longer to close than other issues, with a higher occurrence of technical debts related to code complexity and architecture.

Table 2.1 shows that most empirical studies about code smell and complexity evolution analyze exclusively Java projects. Few studies evaluate other languages or more than one language simultaneously. Our research complements the literature by providing a multi-language empirical study about the evolution of complex methods, which is the most diverse in the literature in terms of programming language.

Table 2.1. Summary of studies by target language.

Languages	**Studies**
Java	[8; 9; 13; 15; 20; 21; 34; 41; 49; 50; 51; 52; 54; 56; 72; 73; 79]
JavaScript	[17; 31; 60]
C	[30; 59]
Python	[10; 75]
Java, C++, and C#	[42]
Java and C++	[33]
Smalltalk	[36]
Python, JavaScript, Java, C++, C#	Ours

2.5 Final Remarks

In this chapter, we presented the central topics of this dissertation, with a focus on Cyclomatic Complexity. We also presented the main benefits and challenges of analyzing complexity at the level of methods/functions as well as the relation between Cyclomatic Complexity and code smells. Lastly, we concluded by discussing the related studies.

In the next chapters, we will rely on those concepts to analyze methods and functions with high complexity, evaluating how they evolve and are modified, what operations are performed, and what perceptions developers have about the complexity.

Chapter 3

Study Design

In this chapter, we describe the design of the studies conducted in this master dissertation. Section 3.1 presents the real software systems covered in our study. Then, Section 3.2 details the methodology used to extract and calculate the complexity of all methods of the target projects. Section 3.3 presents characteristics of the extracted complex methods, comparing them to the other methods. In Section 3.4, we describe how we track the history of complex methods with Git code versioning. Then, in Section 3.5, we present the procedures used to apply and analyze the survey results. Section 3.6 describes the six research questions investigated and how we seek to answer them. Finally, Section 3.7 closes this chapter.

3.1 Selecting Software Systems

In this research, we aim to study real-world and relevant software systems that are written in multiple programming languages. We then select the languages: JavaScript, Python, Java, C++, and C#; those languages are among the top-10 most popular in both GitHub[1] and TIOBE[2] rankings. Next, for each language, we select the 10 most popular software systems based on the GitHub star [5; 64], which is a metric largely adopted in the software mining literature as a proxy of popularity. In this process, we took special care to filter out:

- *Non-software projects*: we removed projects such as tutorials, examples, samples, among others, as those are not real software projects.

[1] GitHub ranking: `https://bit.ly/2XHn2PY`
[2] TIOBE ranking: `https://www.tiobe.com/tiobe-index`

Table 3.1. Selected software systems (Size: number of source files).

Language	Systems	Stars	Size	Commits
JavaScript	angular/angular.js, apache/incubator-echarts, FortAwesome/Font-Awesome, lodash/lodash, moment/moment, mrdoob/three.js, mui-org/material-ui, nodejs/node, prettier/prettier, zeit/next.js	56.3K	1.5K	7.4K
Python	django/django, explosion/spaCy, getsentry/sentry, pandas-dev/pandas, pypa/pipenv, python/cpython, scikit-learn/scikit-learn, ytdl-org/youtube-dl, zulip/zulip, ansible/ansible	29.9K	1.1K	27.4K
Java	apache/dubbo, bumptech/glide, elastic/elasticsearch, google/guava, ReactiveX/RxJava, spring-projects/spring-boot, spring-projects/spring-framework, skylot/jadx, alibaba/fastjson, alibaba/druid	36.1K	3.0K	5.5K
C++	apple/swift, v8/v8, cocos2d/cocos2d-x, emscripten-core/emscripten, facebook/folly, facebook/rocksdb, grpc/grpc, microsoft/terminal, rethinkdb/rethinkdb, tesseract-ocr/tesseract	21.8K	1.7K	27.4K
C#	NancyFx/Nancy, PowerShell/PowerShell, ShareX/ShareX, AutoMapper/AutoMapper, dotnet/efcore, dotnet/wpf, IdentityServer/IdentityServer4, aspnetboilerplate/aspnetboilerplate, dotnet/orleans, icsharpcode/ILSpy	8.1K	1.1K	5.8K

- *Small projects*: we removed projects with less than 500 source files to avoid small and immature ones.

- *Projects with a potential loss of code history*: we also removed projects in which the development did not start in GitHub or were not properly migrated to it. That is, we inspected the very first commit of each candidate project and assessed their source files. We were conservative: we filtered out the projects with more than 10 source files in the very first commit. As we assess software evolution and

perform code history analysis, this is an important threat to be addressed. Thus, by doing so, we ensure we discard projects with a potential loss of code history and keep only the ones that have their full history available.

The 50 selected software systems are presented in Table 3.1. Notice that they include systems that are broadly adopted worldwide, such as Elasticsearch, SpringBoot, Django, Node, Angular, and Swift, to name a few.

The most popular projects are written in JavaScript (median 56.3K stars), followed by Java (median 36.1K stars) and Python (median 29.9K stars); the most popular project is node-js/node with 72.9K stars. Their median size ranges from 1.1K source files in Python to 3K source files in Java. We also verify their activity level: systems written in C++ and Python have the most changes (median 27.4K commits), while Java projects have fewer changes (median 5.5K commits).

Our dataset is publicly available at `http://doi.org/10.5281/zenodo.4546360`.

3.2 Extracting Methods and Computing Complexity

Next, we extract methods from the selected systems and compute their complexity. We use the metric Cyclomatic Complexity, which assesses code branches (*e.g.,*`if`, `for`, `while`, `catch`, `case`, etc) to provide a proxy of complexity. When calculating complexity, logical operators are also taken into consideration (*e.g.,* `&&` and `||`). The rationale is that the more branches a code has, the harder it is to understand and maintain the code. We rely on the open-source tool Lizard[3] to compute Cyclomatic Complexity for multiple programming languages. This tool is also adopted in the software industry, for example, SonarQube[4] and fastlane[5] have plugins for the Apple Swift language that relies on Lizard.

We compute complexity for the methods (and functions) of the selected systems in their latest version. Functions may be considered because programs written in Python and JavaScript are not necessarily object-oriented (for simplicity, we refer to the analyzed entities as methods). For each language, we rank the methods according to their complexity and select the top-200 most complex considering the following filtering criteria:

[3]Lizard project: `http://www.lizard.ws` and `https://github.com/terryyin/lizard`

[4]Popular code quality and security platform: `https://www.sonarqube.org`

[5]Popular tool to automate building and releasing in iOS and Android apps: `https://fastlane.tools` and `https://github.com/fastlane/fastlane`

- *Inactive methods:* as we aim to study code evolution, we defined a minimum number of changes (commits) for the method selection. This way, we discarded methods with three or fewer changes over history. This ensures we assess complex methods that change over time, while we avoid the less active ones.

- *Auto-generated methods:* we manually removed auto-generated files to ensure that the studied methods are created and modified by developers themselves. A file is classified as auto-generated if its source code has an explicit description or comment stating it. Here, we identified and filtered out four auto-generated files.

- *Methods with loss of history:* we also took special care to filter out methods that were renamed or moved, which could cause loss of their code history, biasing our analysis. A possible solution to this problem is to rely on refactoring tools [23; 61; 62; 63; 71] to detect refactoring operations, such as rename/move method. However, this solution is not feasible as those tools mostly focus on the Java language, whereas our work is multi-language. To overcome this threat, we manually inspected the very first commit of each candidate complex method to detect whether it was a commit creating the method or renaming/moving the method. To support this analysis, we rely on the git diff tools. For example, consider the complex method `__call__()` provided by project spaCy: its very first commit[6] is indeed creating the method, thus, this method is included in the analysis. In contrast, the complex method `getBuiltinValueDecl()` provided by project Swift is not included in the analysis because its very first commit[7] is a renaming. Finally, following the guidelines of prior research [59], in the case the candidate method was renamed/moved, we excluded it from the analysis.

It is important to recall that the three aforementioned threats are typically *not* addressed by related studies. That is, commonly, critical issues like auto-generation and renaming are simply ignored. We understand, however, that those threats are fundamental to be addressed to avoid bias in evolutionary analysis, this way, we strive to detect and filter them. This way, after following those filtering criteria, we selected 1,000 complex methods (200 per language).

[6]Commit available at: `https://bit.ly/2LFeFio`
[7]Commit available at: `https://bit.ly/2ZiYDTt`

3.3 Exploring the Complex Methods

Figure 3.1 presents the distribution of the complexity for the top-200 most complex methods, per language. For comparison, we also show the complexity of 200 randomly selected methods, per language. We observe that, independently of the language, the top-200 methods are fairly more complex than the regular methods (all the differences are statistically significant for the Mann-Whitney test, with moderate and large effects for the Cohen effect size). The complexity of the top-200 methods ranges from 22 in JavaScript to 60 in C++, on the median. When considering all languages (last boxplot), the 1,000 selected complex methods have median complexity of 31 (the 3rd quartile is 47).

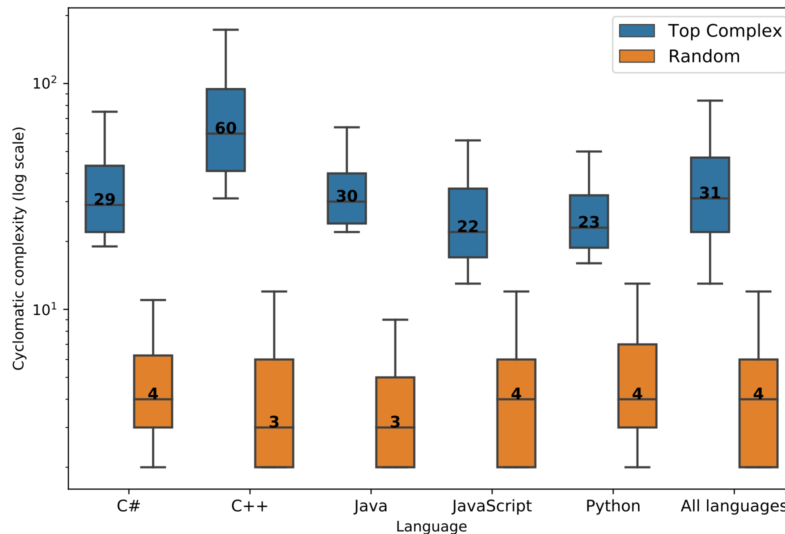


Figure 3.1. Distribution of the complexity of the selected methods.

Method size is a factor that may impact complexity, that is, the longer the method, the more complex it can be. To investigate it, we computed the correlation between complexity and the number of lines of code (NLOC) for the selected methods. Table 3.2 shows this correlation degree using Pearson's Coefficient. Indeed, we notice a moderate and high correlation degree between the two variables in all languages. Therefore, to assess the weight of NLOC to complexity, we compute the ratio between complexity and NLOC for the selected methods and the random ones. Figure 3.2 shows the distribution of these values: even weighing by NLOC, the selected methods continue to be more complex than the random ones per line of code. Again, all the differences are statistically significant, with moderate and large effects.

Table 3.2. Correlation between complexity and NLOC.

Language	Coefficient	Degree
C#	0.749	High
C++	0.846	High
Java	0.623	Moderate
JavaScript	0.548	Moderate
Python	0.853	High
All	0.848	High

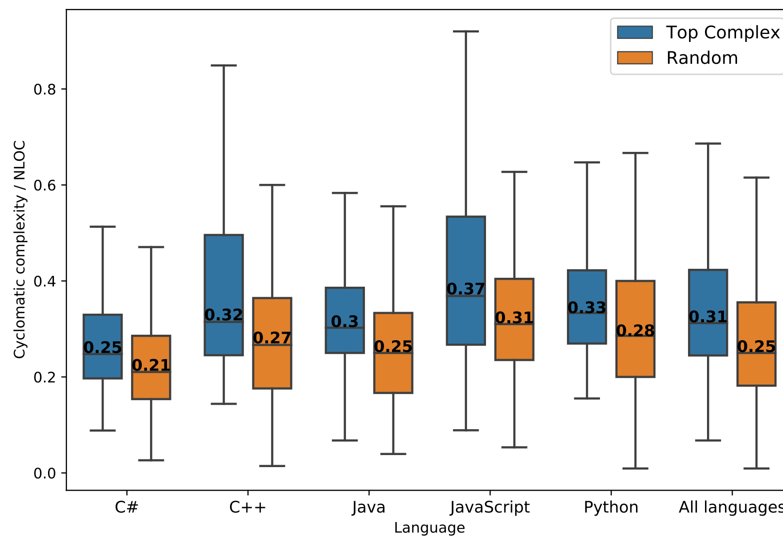


Figure 3.2. Distribution of the complexity per NLOC of the selected methods.

To gain more insights about into dataset, we compute the changeability of the selected methods. Figure 3.3 presents the distribution of commits changing the complex methods (for comparison, the same random methods are used). Indeed, independently of the language, the complex methods have more changes over time. When considering all languages, we notice the frequency of changes 9 for complex methods and 5 for regular ones (all differences are statistically significant, with moderate and large effects).

Finally, Table 3.3 presents the most complex methods for each language. Method `AssembleArchInstruction()`[8] provided by project v8 in C++ is the most complex (627): it contains a long `switch` with several `case` statements and multiple nested `if-else`; the method is responsible for producing machine code and allocating records.

[8]Method available at: `https://bit.ly/34D27kK`

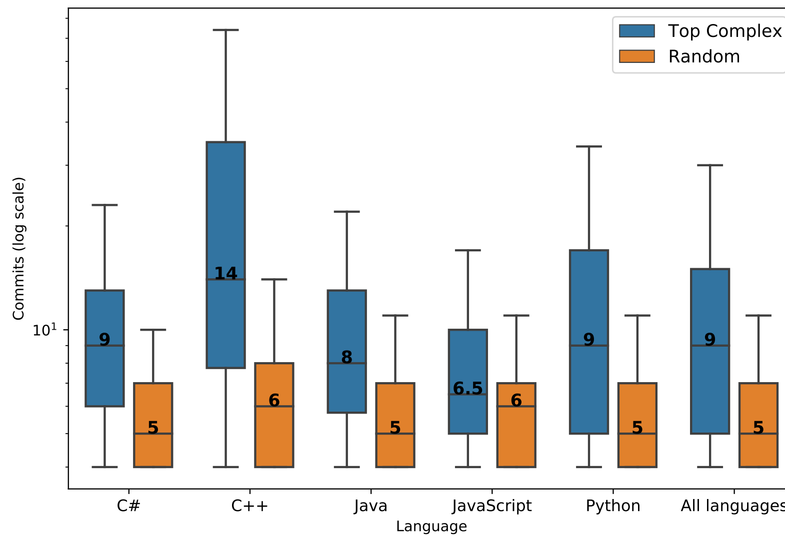


Figure 3.3. Distribution of commits of the selected methods.

The next two complex methods are provided by Elasticsearch in Java and ILSpy in C#. The list is completed with methods of Ansible (Python) and three.js (JavaScript).

Table 3.3. Most complex methods per language.

Name	Project	Lang	Complexity
AssembleArchInstruction()	v8	C++	627
getLegalCast()	Elasticsearch	Java	298
DecodeInstruction()	ILSpy	C#	220
get_virtual_facts()	Ansible	Python	108
parsePathNode()	three.js	JavaScript	80

3.4 Assessing Evolution of Complex Methods

After selecting the most complex methods, we assess their evolution. Specifically, for each complex method, we analyze the commit history of the file including the method with the git command `git log -first parent`, which allows evolutionary analysis of a particular tracked file.[9] Notice that we work at the method level (not at the file level), thus, *all analysis is actually performed at the method level* with the support of the PyDriller mining tool [68].

[9] `https://git-scm.com/docs/git-log#Documentation/git-log.txt`

In this analysis, one threat may exist: the file containing the method may be renamed/moved causing loss of its history. To keep track of file renaming/moving, we use the git option `-follow`; as stated in the git documentation, this option: "*Continue listing the history of a file beyond renames (studies only for a single file)*". This way, we have access to the code history of a file even when it is renamed/moved by developers. This solution is typically adopted by software mining studies to keep track of files over time (*e.g.,* [2; 3]).

3.5 Survey Analysis

3.5.1 Data Collection

In addition to the empirical analysis, we also perform a survey study. At this stage, we aim to study the developers' perceptions of methods considered complex in multiple programming languages. To be as real as possible, *we only focus on developers who have actually maintained complex methods in the selected projects.*

To find the candidate developers, we start by collecting the latest modification of all methods with 10 or more complexity. We then select the 1,000 most recent modifications and collect the developer (name and email) who performed the modification. We select the most recent changes to ensure developers are more likely to remember them. In this process, we remove methods that are changed by the same developer, *i.e.,* modifications with the same email. Thus, with the support of Git, we collect data about 1,000 distinct developers and their corresponding modifications on complex methods.

We elaborate three questions for those developers, considering (1) the impact of complexity on maintenance tasks, (2) the difficulty to maintain complex methods, and (3) the reasons for the existence of complex methods. We also include information about the modified complex method, such as its name, its filename, and the commit URL to GitHub. This information can help the developers identifying and remembering the changes. Specifically, we sent the following email:

Dear [developer],

I am a researcher working on software maintenance and code smells.
In my research, I am studying the complex methods of [project_name].

I found that you changed the following complex method in this project:

- Method: [method_name]
- File: [file_name]
- Commit link: [commit_url]

Could you please answer the following three questions about this commit:

1) Do you think this complex method is harmful for maintenance?

2) Was this change somehow harder to implement since it happened inside a complex method? Why?

3) Why do you think developers have never refactored such a complex method before?

Please, provide any other comments or thoughts regarding the maintenance of complex methods.

We sent 1,000 emails during one month, that is, 50 emails per weekday. For this purpose, we rely on the GMass tool[10], which automates the task of sending emails by generating customizable emails. Considering the 1,000 sent emails, 63 did not find the recipient (*i.e.,* the domain or email are invalid). We received answers from 73 developers, thus, achieving a response rate of 7.8% (73/937). Table 3.4 presents the number of developers who responded to the survey per language and project. The language with the most answers is C++ (47%), while the one with the fewest answers is C# (5%). Notice that responses come from developers who modified complex methods in highly relevant projects, including Apple Swift, Django, CPython, Angular.js, and Guava. Furthermore, some of those developers come from large and worldwide companies, such as Google, Facebook, Apple, Microsoft, and Oracle.

[10] https://www.gmass.co

Table 3.4. Overview of the respondent developers.

Language	Systems	Devs	%
C++	apple/swift, v8/v8, facebook/folly, facebook/-rocksdb, grpc/grpc, microsoft/terminal, rethinkdb/rethinkdb, tesseract-ocr/tesseract	34	47
Python	django/django, getsentry/sentry, python/-cpython, scikit-learn/scikit-learn, ytdl-org/-youtube-dl, zulip/zulip, ansible/ansible	20	27
JavaScript	angular/angular.js, mui-org/material-ui, nodejs/-node, prettier/prettier	08	11
Java	ReactiveX/RxJava, google/guava, spring--projects/spring-framework, elastic/elasticsearch	07	10
C#	PowerShell/PowerShell, dotnet/wpf, dotnet/-orleans	04	5

3.5.2 Data Analysis

Figure 3.4 presents an overview of the developers' responses: it shows a word cloud of the 50 most used words in their responses. Larger words mean they are more frequent, while smaller words represent less frequent ones. The most frequent word is *refactor*, followed by *hard* and *test*. Other words are also highlighted, including *maintenance*, *difficult*, and *easy*. While the word cloud provides an overview of the responses, it does not identify the context in which the words are used.

Figure 3.4. Word cloud of the developers' responses.

As a way to analyze the developers' responses quantitatively, we apply *thematic analysis* [14] on each question, similar to previous studies [7; 70]. The analysis seeks to

identify and record themes in textual documents, using the following steps: (1) initial reading of the responses, (2) generating a first code for each response, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes. The first three steps were carried out by the first author of the dissertation, while steps 4 and 5 were developed by the consensus of the two authors through several meetings and discussions. As the responses are in open fields and not mandatory, there are cases in which the respondents only answered one question, or answered all questions in a single paragraph. Thus, manual analysis was needed in few cases to link questions and responses.

3.6 Research Questions

The first part of this study provides a quantitative analysis of the evolution of complex methods (RQs1-3). To gain more insights, we provide a qualitative analysis of the developers' perceptions of complexity (RQs4-5). Next, we detail how we analyze each research question.

RQ1: How do complex methods evolve over time?

In the first research question, we assess how changes performed by developers affect the complexity of the selected methods. For each method, we compute its complexity over time and assess whether it is more likely to increase or decrease complexity. This classification is obtained by applying the Mann-Kendall trend test [32; 45]. For this purpose, we compute the complexity of each method per week in a time series, so the test can identify the trend of the data. In this analysis, we only consider methods with 10 or more weeks between the first and last changes to ensure a more precise result [80]; we only discarded 14 methods with less than 10 weeks.

Methods that tend to become even more complex during evolution are harmful to maintenance and should be addressed with care. Likewise, efforts to decrease complexity at method level can be detected and fomented by development teams. So far, it is not clear the impact of the programming language on the evolution of complex methods.

RQ2: What changes are performed on complex methods?

We now focus on the reasons that lead the complex method to be changed. We then inspect the commit messages of all methods (complex and non-complex) and search for keywords describing the change goal. Particularly, we classify the commits into three

change types: *bug*, *new feature*, and *refactoring*. These three categories summarize the major activities in software development, such as fixing bugs, adding new features, and refactoring code, and is largely adopted in the literature as a proxy to classify commit changes [1; 4; 19; 24; 35; 58; 69; 81]. For bug, we search for commit messages with the keywords *bug*, *fix*, or *patch* [1; 24; 35; 81]. For new feature, we consider the keywords *new feature* and *add* [4]. Lastly, for refactoring, we consider the keywords *refactor* and *code clean* [4; 58; 69]. Notice that a commit can also be classified into two or three categories since it can contain two or more keywords of distinct change types; this may happen, for example, when unrelated code changes are committed together (*e.g.,* bug-fix and refactoring), creating the so-called *tangled commits* [16].

To validate this classification, we perform a manual evaluation. We randomly select 384 commits (*i.e.,* 95% confidence level and 5% confidence interval) and manually inspect them. We then carefully verified whether the automated classification was correct by manually reading the commit messages as well as the code changes made in the commit. This assessment leads to a fair precision of 81%, meaning the automated classification was able to correctly flag the change type in over 4 out of 5 commits.

Finally, the extracted data includes two types of methods (complex and non-complex) and three types of changes (bug, new feature, and refactoring). We then assess whether these two variables (method and change type) are related or independent. For this purpose, we rely on the Chi-Squared test, which is adopted when there are two categorical variables, each with two or more possible values. After applying this test, we assess the Pearson residuals to measure the difference between the observed and expected frequencies. When the absolute value of the residual is greater than two, we consider that the observed frequency is significantly higher than the expected [11; 26]. Thus, we can verify whether complex methods have a higher/lower concentration of bugs, new features, and refactoring as compared to not-complex methods.

We aim to assess whether changes in complex methods are any different from regular methods and whether programming languages play a role in this analysis. We also aim to explore which changes are more and less common. Identifying what changes are more likely to happen in complex methods and what languages are more harmful is important to bring to light data to better support maintenance efforts.

RQ3: What operations are implemented in complex methods and which ones are more likely to become more complex over time?

In this research question, we explore the content of the complex methods. As a solution to automate this analysis, we focus on the method prefixes, which are very often

verbs, and indicate the kind of operations they perform [27; 29; 46]. For instance, methods prefixed with *create*, *builder*, and *generate* may indicate *creation* methods, while methods prefixed with *convert*, *format*, and *to* may indicate *conversion* methods. We initially rely on a previous list of prefixes and their corresponding categories to infer the operations of the complex methods [27]. Prefixes not present in this previous list were manually classified by the authors of this dissertation.

We aim to assess whether complex methods are over/under concentrated on certain operations. Also, we can better understand which operations are more/less likely to be problematic by assessing their complexity over time. This can provide the basis for the creation of more focused tools to spot complex methods and prioritize maintenance efforts, not only according to complexity but also according to their operations.

RQ4: What are developers' perceptions of method complexity?

In this qualitative analysis, we seek to understand the impact of complexity on developers' maintenance tasks. For this purpose, we assess whether complex methods are harmful to software maintenance. We also analyze, based on the developers' perceptions, whether changes are somehow harder to be implemented when they happen in complex methods. To answer this research question, we analyze Questions 1 and 2 of the proposed survey with the support of thematic analysis.

RQ5: Why complex methods are not eliminated from code?

Despite the drawbacks of code complexity are well-known [22; 46; 70], complex code is present in most real software projects. We, therefore, aim to understand the reasons for the existence of complex methods. For this purpose, we assess why developers continue to create complex methods over and over again and why there is no large effort to handle complex methods. To answer this RQ, we assess Question 3 of the proposed survey with the support of thematic analysis.

RQ6: To what extent are self-admitted complex methods different from other complex methods?

We now seek to better understand what are the main differences between self-admitted complex and self-admitted not complex methods according to the perception of the developers in the survey. For this, we analyzed three distinct aspects of the self-admitted methods:

- **Developer Experience.** To assess this characteristic, we need to verify the developers' experience. In this context, as a proxy of experience, we compute the *number of commits* (NOC) of the developer in the project. Specifically, for each project, we compute the distribution of the number of commits of its developers. Then, we rely on the values of the first and third quartiles to classify the level of commits of the surveyed developers as compared to the other developers in the same project. We classify the surveyed developers in three categories of experience:
  - *Low experience*: NOC < first quartile
  - *Medium experience*: first quartile ≤ NOC ≤ third quartile
  - *High experience*: NOC > third quartile
- **Code and Evolution.** We aim to explore the code and evolutionary differences between the two groups of complex methods. For this, we extract and compare some software quality metrics, namely: (1) Cyclomatic Complexity; (2) Lines of Code; (3) Tokens (*i.e.,* number of words and operators); (4) Number of Parameters; (5) Commits (*i.e.,* number of commits that changed the target method); (6) Growth Rate (*i.e.,* the current complexity of the method divided by the complexity of its very first version). The first four metrics evaluate aspects of the source code, while the last two verify its evolution over time.
- **Maintenance.** We seek to understand how both groups of complex methods affect software maintenance. As a proxy of maintainability, we analyze the issues and pull requests (PRs) related to each complex method. First, based on that data, we extract two metrics: (a) number of messages in issues and pull requests and (b) number of changed files in pull requests. The rationale is that the higher those numbers, the more discussion and effort are dedicated to the proposed changes. Then, to further explore possible maintainability issues in the target methods, we assess the most common maintenance problems on their issues and pull requests. For this purpose, we apply another *thematic analysis*, such as the survey, to identify and record themes.

Prior studies report that there exists a gap between theory and practice in software development: what is believed to be a problem is not always a real problem for developers [53]. For example, some code smells [53] and test smells [55] are generally not perceived by developers as design problems. Thus, it is important to assess

self-admitted complex methods because those methods represent the perception of the maintainer developer rather than any code metric.

3.7 Final Remarks

This chapter presented the design of the experiments reported in this master dissertation. First, we described the selection of analyzes software systems. We selected 50 projects according to the number of stars in the GitHub of the five programming language covered: C#, C++, Java, JavaScript, and Python. Then, we searched and extracted the 1,000 most complex methods from these projects, comparing them with the other methods. We also presented how we build and applied the survey with real developers to understand the developer's perception of complex methods. We received more than 70 responses and relied on thematic analysis to extract the main themes mentioned. Finally, we described how we assess the six research questions.

The next three chapters present and discuss the results of all our research questions, analyzing the implications of each finding and the threats to validity of each chapter. Chapter 4 answers RQ1 to RQ3, while Chapter 5 details RQ4 and RQ5. Lastly, we expose the results of RQ6 in Chapter 6.

Chapter 4

Empirical Results: Evolution of Complex Methods

This chapter provides the empirical results of the first three research questions (RQs1-3), which involve the evolution of complex methods over time. Section 4.1 presents how complex method evolve (RQ1). In Section 4.2, we assess the modifications in the complex methods (RQ2). Section 4.3 presents the results of RQ3, evaluating the main operations implemented in the complex methods. In Section 4.4, we detail findings and implications. Finally, we conclude the chapter by presenting Threats to Validity in Section 4.5 and Final Remarks in Section 4.6.

4.1 RQ1: How do complex methods evolve over time?

In this research question, we assess the evolution of complex methods. Figure 4.1 compares the complexity of the selected methods in their first, intermediate, and last versions. Overall, their complexity tends to increase over time: on the median, from 23 in the first version to 31 in the last one, representing a global gain of about 35% in complexity. The difference between the first and last versions is statistically significant for the Mann-Whitney test, with a moderate effect for the Cohen effect size.

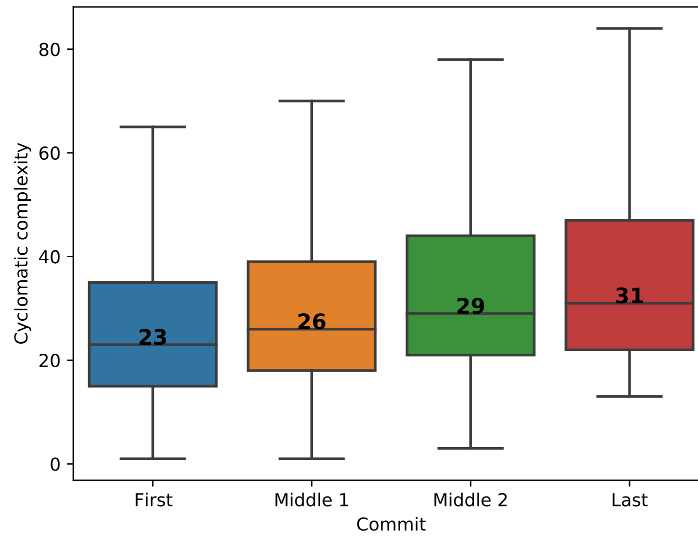


Figure 4.1. Distribution of complexity in the first, intermediate, and last versions.

Trend analysis. We detail in the pie chart of Figure 4.2 the ratio of complex methods that increased and decreased in complexity over time according to the Mann-Kendall trend test. The majority of the target methods increased complexity (red color), independently of the programming language. The proportion of methods that increased complexity ranges from 46.5% in C# to 75.3% in C++. In contrast, only a minority of the methods decreased complexity (green color); in this case, the proportion ranges from 7.6% in C++ and JavaScript to 17.2% in Java. In addition, there are methods in which complexity showed no trend (orange color).

Figure 4.2 also presents on the right side the distribution of commits on each trend category. Methods that increase complexity have 10 commits on the median (the first quartile is 7 and the third quartile is 21), while methods that decrease it have 9 commits (the first quartile is 6 and the third quartile is 12.25). This difference is statistically significant for the Mann-Whitney test, with a moderate effect for the Cohen effect size. Interestingly, the methods with more changes are the ones that tend to become more complex over time. In contrast, the methods with fewer changes tend to have less degradation. Figure 4.3 presents examples for each category. On the left, we see a method increasing complexity: method `AssembleArchInstruction()` provided by project V8 has changed close to 300 times and increased its complexity to over 400. In the center, we present an example of decreasing complexity: method `createProxy()` of Apache Dubbo has close to 50 commits and its complexity has drastically decayed.

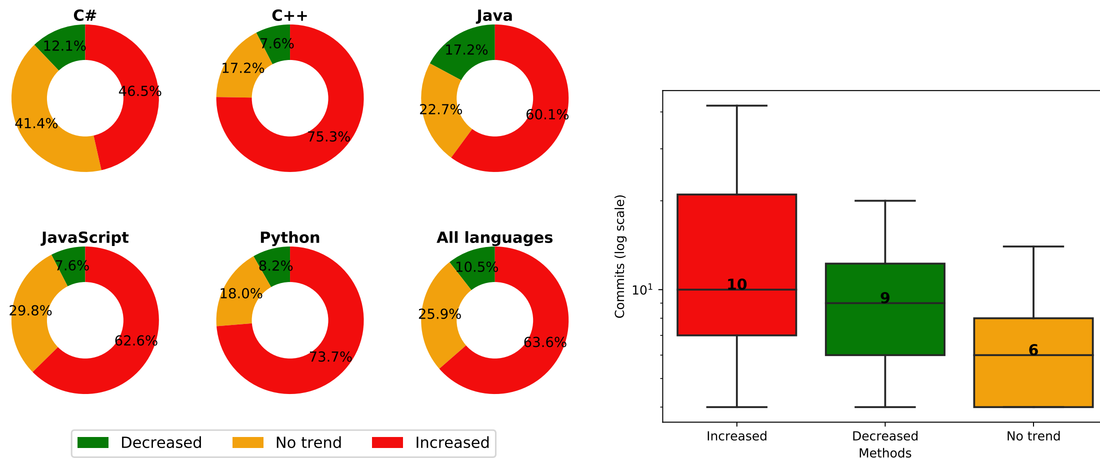


Figure 4.2. Trend analysis. (left): proportion trend categories. (right): distribution of commits per trend category

Finally, on the right, we present an example that does not change complexity over time: method `splitToN()` of ElasticSearch remained with complexity 27 after 9 commits.

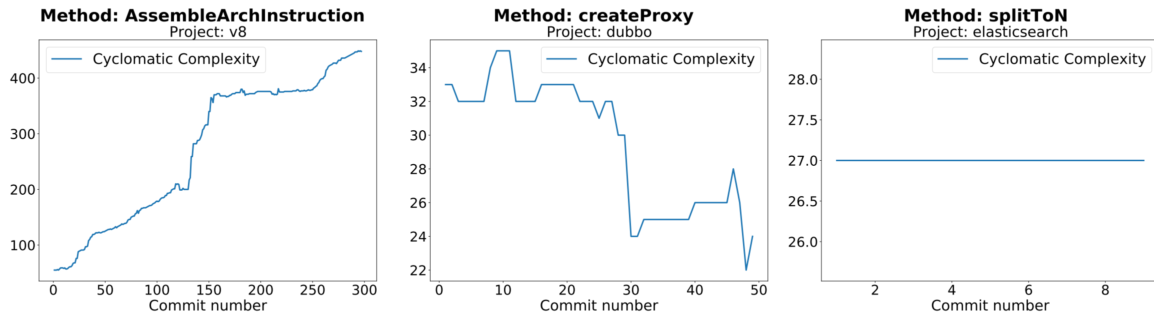


Figure 4.3. Examples of complex methods per category. (left): increasing complexity. (center): decreasing complexity. (right): same complexity.

Assessment of project and code size. We have seen so far that the five studied languages have distinct trends regarding complex methods. For example, C++ tends to have more methods that increase complexity over time, while Java has more methods that decrease complexity. To better understand possible reasons behind that, we now assess two properties of the target projects: project size and code size.

Figure 4.4(a) summarizes the project size by presenting the distribution of the number of source files per project. We see that Java projects have significantly more source files (3,065, on the median) than other programming languages. This larger number of source files in Java may suggest a better distribution of responsibilities. Interestingly, Java is the language with the most complex methods decreasing complexity over time (17.2%) and only the 4th one with methods increasing complexity (60.1%).

If one considers that having more source files implies having focused responsibilities, this may explain why Java has more effort to reduce complexity than other languages. Figure 4.4(b) presents the distribution of the number of lines code per source file. We observe that C++ projects have the largest source files, with a median of 127 lines of code. This shows that having large source files is somehow common practice in this language. In the trend analysis, we have seen that most C++ complex methods tend to increase complexity over time (75.3%), and only 7.6% actually decreased complexity. Thus, the naturally larger size of the source files in C++ may explain why they are more prone to have complex methods and less effort reduce their complexity.

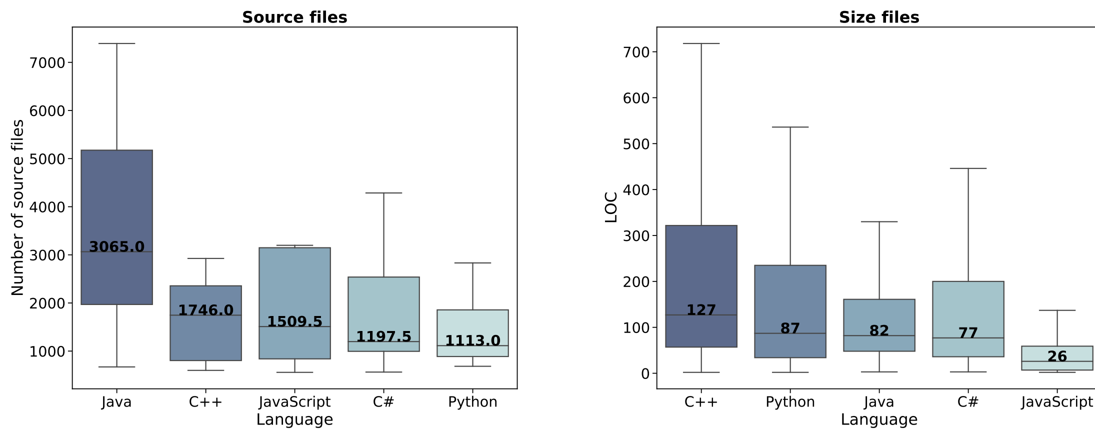


Figure 4.4. Distribution of project size and code size.

Manual inspection of code changes. To better understand the content of those changes, we manually inspected the code changes of methods on each trend category. Specifically, for each category, we randomly selected 10 methods (2 per language, with at least 10 commits), totaling 30 methods. We then manually assessed their 520 commits and inspected their code changes, paying special attention to modification, addition, and removal of the following code elements: method invocation, variable declaration/usage, conditional, loop, and formatting.

Table 4.1 presents the values for this analysis (note that the sum of the percentages may exceed 100% because a commit can involve several code changes, *e.g.,* change code formatting and add a conditional. Overall, changes in conditional structures are more frequent than changes in loop structures, for example, in the *increased* category 53% of the analyzed commits changed the conditionals, while only 13.3% changed loops. As expected, the complex methods classified as *increased* and *decreased* changed more conditional and loop structures than the *no trend*, since the complexity is measured by the number of these structures. In contrast, the *no trend* category showed fairly

more modifications involving code formatting (*e.g.,* line breaks, comments, indentation, etc), which can explain why these methods have changed and yet do not have their complexity increased nor decreased.

Table 4.1. Code changes in complex methods.

Categories	**Invocation**		**Variable**		**Conditional**		**Loop**		**Formatting**	
	#	%	#	%	#	%	#	%	#	%
Increased	144	71.3	130	51	107	53	27	13.3	28	14
Decreased	99	58.2	64	37.6	73	43	12	7	16	9.4
No trend	77	52.0	51	34.4	45	30.4	10	6.7	37	25

Table 4.2 breaks this data per language and presents the most frequent changes. Notice that changes in *conditionals* are more frequent than changes in *loops* in all languages. Moreover, changes in *conditionals* and *loops* are more frequent in C++ than in other programming languages, which may explain why C++ is the language with the highest proportion of methods that increased complexity over time. On the other side, C# has the largest number of changes related to *formatting*. Indeed, this can explain the highest proportion of methods classified as *no trend* (41.4%) in that language.

Table 4.2. Code changes in complex methods per language (%).

#	**C++**		**Python**		**Java**		**C#**		**JavaScript**	
1º	**Cond.**	**32.96**	Invoc.	40.99	Invoc.	36.49	Invoc.	34.23	Invoc.	36.70
2º	Invoc.	32.96	Var.	29.19	Cond.	27.00	Cond.	27.02	Var.	34.17
3º	Var.	25.55	Cond.	19.87	Var.	24.08	**Format.**	**22.52**	Cond.	23.41
4º	**Loop**	**07.41**	Format.	08.69	Format.	11.67	Var.	13.51	Format.	13.92
5º	Format.	01.11	Loop	04.96	Loop	00.72	Loop	02.70	Loop	09.49

Summary of RQ1: Overall, complex methods tend to become even more complex over time, while the effort to decrease their complexity is minimum. However, this varies per programming language: C++ projects have more methods that increase complexity, whereas Java ones present more effort to reduce complexity. The addition of conditional structures (rather than loops) is the major contributor to increase complexity over time.

4.2 RQ2: What changes are performed on complex methods?

Overall analysis. We now assess the motivation behind the changes in the complex methods by inspecting the commit messages. For comparison, we also compute the same data for non-complex methods. Table 4.3 summarizes our categorization for the two types of methods (complex and not complex) and the three types of changes (bug, new feature, and refactoring). Overall, the most common type of change is related to bugs, followed by new features and refactoring. Interestingly, the complex methods have proportionally more changes related to bugs (32.65% vs. 30.36%), new features (15.72% vs. 11.83%), and refactoring (3.69% vs. 1.64%) than the non-complex ones.

Table 4.3. Change classification in complex and not complex methods.

Categories	Bug		New Feature		Refactoring	
	#	%	#	%	#	%
Complex	3,400	32.65	1,635	15.72	384	3.69
Not Complex	348,773	30.36	135,953	11.83	18,887	1.64

Table 4.4 details this analysis per programming language when considering only the complex methods. Notice that C++ and Python have the highest proportion of changes related to bugs and new features, while refactoring is less common. On the other hand, changes related to refactoring are proportionally more common in Java (5.57%) than in other languages. These results complement the findings of RQ1. First, we have seen that C++ and Python complex methods tend to become more complex over time. This may happen due to the addition of new features and bug-fixing, without proper refactoring activities. Second, Java has more methods that decrease complexity, and one explanation is that refactoring is more prone to happen in this language (for example, to keep responsibilities more focused, as presented in RQ1).

Table 4.4. Change classification in complex methods by language.

Cat.	C++		Python		Java		C#		JavaScript	
	#	%	#	%	#	%	#	%	#	%
Bug	2,230	**29.16**	547	**19.34**	251	10.52	163	07.72	209	11.84
New Feat.	877	**11.47**	349	**12.34**	147	06.16	150	07.10	114	06.45
Refactoring	188	01.54	45	01.59	133	**05.57**	42	01.98	66	03.73

Observed and expected frequencies. To better understand this data, we apply the Chi-Squared test and compute the Pearson residuals to measure the difference between

the observed and expected frequencies. We recall that when the absolute value of the residual is greater than two, the observed frequency is considered significantly higher than expected, while values close to 0 indicate that there is no significant difference between them [11; 26]. Table 4.5 shows the residuals for bugs: the condition *complex* and *with bug* is over-represented (> 2), while the condition *complex* and *without bug* is under-represented (< 2). This reinforces the idea that complex methods significantly undergo more changes related to bug fixes than less complex methods.

Table 4.5. Residuals for bugs.

	with bug	without bug
Complex	4.20	-2.78
Not Complex	-0.40	0.26

Table 4.6 and 4.7 present the residuals for the new feature and refactoring changes. We see that both conditions *complex* and *with new feature* as well as *complex* and *with refactoring* are also over-represented (> 2). Thus, complex methods significantly have more changes related to new features and refactoring than less complex ones. Finally, it is worth noticing that refactoring changes are the ones most over-represented in complex methods (16.03), followed by new features (11.41) and bugs (4.20).

Table 4.6. Residuals for new features.

	with new feature	without new feature
Complex	11.41	-4.19
Not Complex	-1.09	0.40

Table 4.7. Residuals for refactoring.

	with refactoring	without refactoring
Complex	16.03	-2.08
Not Complex	-1.53	0.20

Summary of RQ2: Complex methods are significantly over-represented on changes related to bugs, new features, and refactoring as compared to non-complex methods. Overall, refactoring presents the highest observed frequency: complex methods undergo 2.2x more refactoring than not complex ones (*i.e.,* 3.69% against 1.64%). There is some distinction per language: C++ and Python complex methods contain more changes related to bugs and new features, while Java has more changes related to refactoring.

4.3 RQ3: What operations are implemented in complex methods? Which ones are more likely to become more complex?

In this research question, we explore the content of the complex methods. To automate content discovery, we use the method prefixes to infer the operations they perform [27; 29; 46]. As explained in Chapter 3, we rely on a previous list of prefixes and their corresponding categories to infer the content of the complex methods [27]; when a prefix is not present in this list, we perform a manual classification. Table 4.8 presents the 13 content categories as well as examples of prefixes and methods on each category. For instance, the category *processing* include methods prefixed with *process*, *extract*, *calculate*, among others.

Table 4.8. Content categories of the complex methods.

Categories	**Prefix examples**	**Method example**
Accessing	get, set, value	getLegalCast()
Collection	collect, visit, next	nextToken()
Condition	is, has, can	HasChildItems()
Conversion	convert, format, to	convertBuiltinType()
Coordination	schedule, update, wait	update_groups()
Creation	create, builder, new	buildTable()
IO	read, write, save	read_setup_file()
Processing	process, extract, calculate	ProcessRecord()
Release	release, clear, shutdown	clearBuffer()
Setup	setup, configure, init	loadSchema()
Test	test, assert, mock	testRandomDecision()
Validation	verify, check, validate	verifyFile()
Undefined	-	yylex()

Content analysis. Table 4.9 presents the results of this classification. The first column shows the number of complex methods in each category in absolute and relative terms. For example, we find 335 complex methods (33.5%) in category *processing*. Column "All Methods" presents the overall presence of each category as measured by hundreds of Java systems [27]; for example, *processing* is present on 4.87% methods when considering all methods. Lastly, column "Proportion" shows whether the category is more represented in the complex methods (ratio > 1) or in all methods (ratio < 1).

Considering the category *processing*, we notice a ratio of 6.88 (*i.e.,* 33.5/4.87), meaning that this category is proportionally much more concentrated in the complex

Table 4.9. Concentration of complex methods per content category. Concentration column: Low: ratio < 1; Medium: $1 \leq$ ratio < 2; High: ratio ≥ 2. *Frequency of "All Methods" is based on [27].

Categories	Complex Mtds		All Mtds*	Proportion (Complex/All)	
	#	%	%	Ratio	Concent.
Processing	335	33.5	4.87	6.88	High
Conversion	68	6.8	2.68	2.53	High
Validation	32	3.2	1.28	2.50	High
IO	78	7.8	3.89	2.01	High
Setup	35	3.5	1.81	1.93	Medium
Creation	62	6.2	4.50	1.38	Medium
Collection	62	6.2	4.57	1.36	Medium
Coordination	48	4.8	3.94	1.22	Medium
Condition	49	4.9	5.53	0.89	Low
Release	7	0.7	1.38	0.51	Low
Accessing	115	11.5	22.87	0.50	Low
Test	40	4.0	8.22	0.49	Low

methods than in all methods. In addition to *processing*, the complex methods are over-concentrated on *conversion*, *validation*, and *IO*. On the opposite side, some categories are under-represented in the complex methods (*i.e.,* ratio < 1). For instance, although *accessing* methods are the most common category in all methods (22.87%), it is present only in 11.5% of the complex methods, having a ratio of 0.50. In addition, the categories *condition*, *release*, and *test* are also under-represented in the complex methods.

Table 4.10 details the top-3 most frequent categories per language. Independently of the programming language, the most adopted method category is *processing*, with a higher concentration in Python. The other categories slightly vary per language, however, they mostly concur with the overall analysis presented in Table 4.9.

Table 4.10. Categories of the complex methods by language.

#	C++		Python		Java		C#		JavaScript	
1º	Proc.	06.67	Proc.	09.34	Proc.	06.57	Proc.	05.85	Proc.	06.57
2º	Crea.	02.44	Valid.	02.34	IO	01.92	Setup	03.03	Conv.	04.85
3º	Conv.	02.05	Coord.	02.03	Collec.	01.64	IO	02.44	Valid.	03.51

Content evolution. Next, we assess how each category changes over time (Table 4.11). For each category, we present the number of commits as well as the complexity in the first and last versions (median values). We also present the growth of complexity as measured by the ratio last/first. For instance, *collection* methods have 9

commits and a complexity of 21.5 in the first version, growing to 36.5 in the last version (+70%). In addition to *collection* methods, two categories have growth higher than 40%: *conversion* (+47%) and *coordination* (+44%). Observe that these three fastest-growing categories represent together only 17.8% of the methods (see Table 4.9). That is, although *collection*, *conversion*, and *coordination* methods are not very prevalent in our dataset, they are among those that become more complex over time. Notice that other categories have low growth in complexity, particularly, *setup* and *release*. Interestingly, despite being the most changed with 18 commits, *release* methods have the lowest growth in complexity (+7%), from 15 to 16.

Table 4.11. Growth of complex methods per category. Growth column: Low: ratio < 1.2; Medium: $1.2 \leq$ ratio < 1.4; High: $1.4 \geq$ ratio. Commits, first, and last are median values.

Categories	**Commits**	**Complexity**		**Proportion (Last/First)**	
		First	**Last**	**Ratio**	**Growth**
Collection	9.0	21.5	36.5	1.70	High
Conversion	7.0	23.5	34.5	1.47	High
Coordination	8.5	21.5	31.0	1.44	High
Accessing	10.0	25.0	34.0	1.36	Medium
IO	7.0	23.5	31.5	1.34	Medium
Processing	8.0	22.0	29.0	1.32	Medium
Condition	9.0	25.0	32.0	1.28	Medium
Test	7.0	20.5	26.0	1.27	Medium
Creation	10.5	31.0	38.0	1.23	Medium
Validation	8.0	21.5	26.0	1.21	Medium
Setup	9.0	19.0	22.0	1.16	Low
Release	18.0	15.0	16.0	1.07	Low

To illustrate how complex methods evolve, we provide a detailed analysis for the categories with the highest growth in complexity (*i.e.*, *collection*, *conversion*, *coordination*, and *accessing*). For each category, we present and discuss a method in its first and last commit. To facilitate code reading and understanding, we annotate the examples as follows:

- *Green arrows* represent code elements presented in the last version but not in the first version.
- *Red arrows* represent code elements presented in both the first and the last versions.
- *Three dots* represent omitted code with no impact on the complexity.

Collection. This category has the highest growth in complexity when comparing the first and last method versions (+70%). Methods in this category are responsible for managing or iterating in collections of objects. They contain prefixes like *collect*, *visit*, *add*, and *next*, for example, `AddTraceListenersToSources()`, `SelectProductName-ForDirectory()`, and `VisitMethodCall()`. Figure 4.5 presents the method `Visit-InvocationExpression()` of project ILSpy (C#), which is part a Visitor design pattern implementation. This method was initially less complex, but became significantly more complex: over time, its complexity has increased from 10 to 20 in 9 commits. Most of the new conditionals are type checks, possibly to fix bugs and handle new behavior. For this purpose, the `is` operator[1] is largely adopted in the new version. Notice that the over-usage of type checking is considered a symptom of poor code [22].

Conversion. This category includes methods that have the role of converting data types. Conversion includes methods with prefixes like *convert*, *format*, *from*, and *to*. Examples of methods in this category include: `convertBuiltinType()`, `Convert-Symbol()`, and `transformOracleToPostgresql()`. As an example, Figure 4.6 presents the method `dayOfYearFromWeekInfo()` of project Moment (JavaScript), which is responsible for date conversion. In the first version, we see a smaller and less complex method, with only four `if` statements. However, over time, several checks were added to verify the limits of the manipulated variables (*i.e.,* `week` and `weekday`) and handle possible input errors, thus, its complexity increased from 7 to 15. These novel checks are all mixed with the old code, generating a code that is harder to follow due to the excessive number of branches.

Coordination. This is the third category with the highest growth in complexity and it is related to task coordination, such as sending data and event notifications. Methods in this category have prefixes like *schedule*, *update*, *try*, and *start*, for example, `update_groups()`, `StartRecording()`, and `TrySetBreakpoint()`. Figure 4.7 presents the method `handle()` of ElasticSearch (Java), which handles requests. This method has tripled its complexity over 5 commits, increasing its complexity from 7 to 22. Several conditional statements were added over time to handle user authentication and provide security to the users. Those novel checks are mixed with the old code but also stacked on the bottom of the method to cover each new case.

Accessing. Methods in this category have prefixes like *get* and *set*, for example, `get_attrs()` and `SetItem()`. They have the role of accessing private object information, assisting encapsulation in object-oriented programming. It is important to notice that, those methods should ideally be simple and small because their sole goal

[1] `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/is`

Figure 4.5. Collection method example (project ILSpy).

is accessing certain data. However, our analysis reveals that accessing methods are the fourth category with the highest growth in complexity (+36%). Figure 4.8 shows

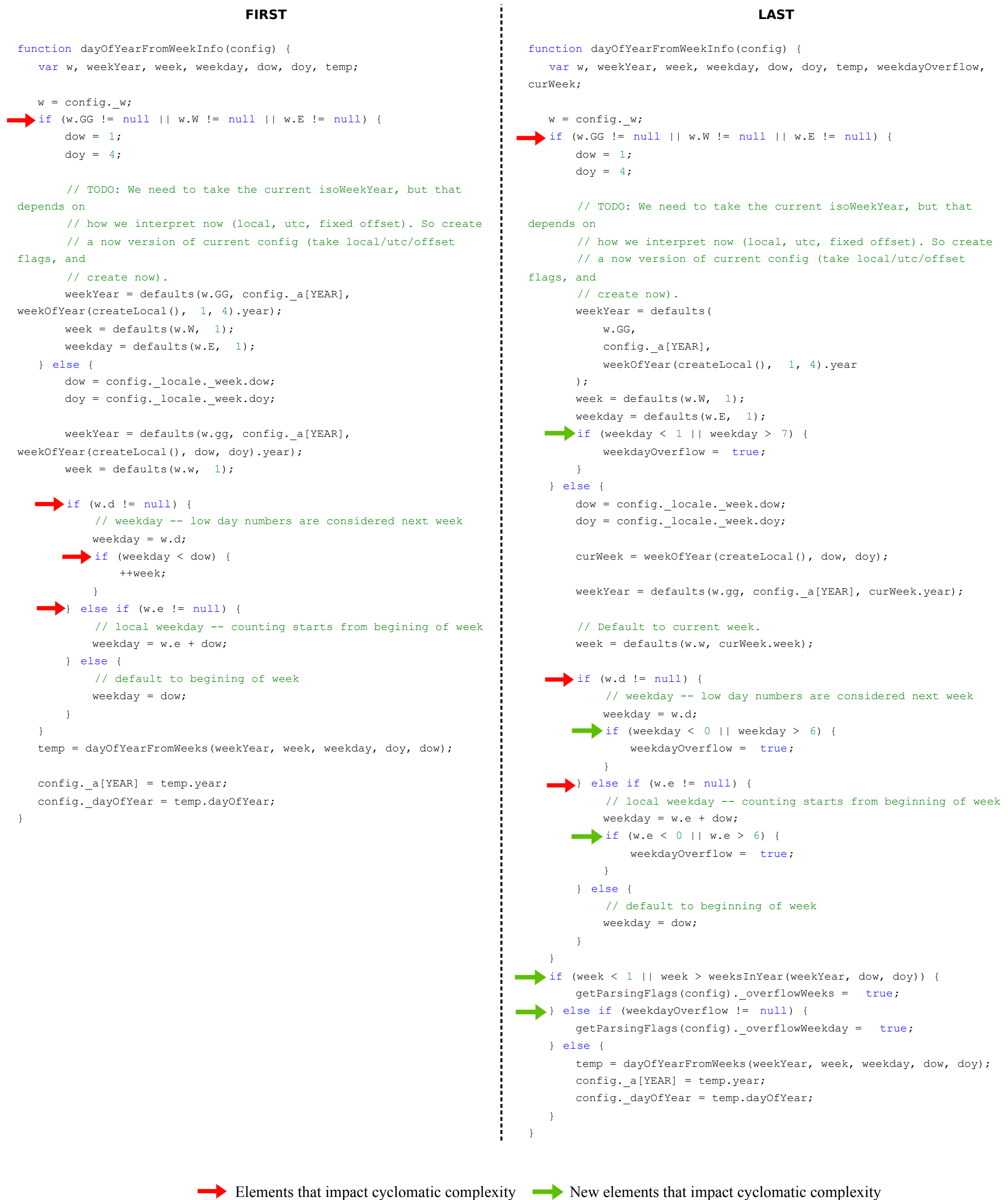


Figure 4.6. Conversion method example (project Moment).

the method `GetWeightedValues` of project Nancy (C#). This method has completely changed over time due to the addition of lazy initialization, therefore, its complexity skyrocket, from 3 to 25. As result, the method is no more a simple getter, but it also handles cache and initialization details.

Figure 4.7. Coordination method example (project ElasticSearch).

Summary of RQ3: Complex methods are not equally distributed among different operations. On the contrary, complex methods are clearly over-represented on certain operations, namely *processing*, *conversion*, *validation*, and *IO*. The growth in complexity varies according to the operations: *collection*, *conversion*, and *coordination* methods are more critical because they present a faster growth in complexity, whereas *setup* and *release* methods have a slower growth.

FIRST

```
private IEnumerable<Tuple<string, decimal>> GetWeightedValues( string
headerName)
{
    var values = this.GetSplitValues(headerName);
    var parsed = values.Select(x => {
        var q = x.Split( new[] {";q="},
StringSplitOptions.RemoveEmptyEntries);
        var quality = 1m;
        if (q.Length > 1){
            if(!decimal.TryParse(q[ 1], NumberStyles.Float,
CultureInfo.InvariantCulture,  out quality)){
                return null;
            }
        }
        return new Tuple<string, decimal>(q[0].Trim(), quality);
    });

    return parsed
        .Where(x => x !=  null)
        .OrderByDescending(x => x.Item2);
}
```

LAST

```
private IEnumerable<Tuple<string, decimal>> GetWeightedValues( string
headerName)
{
    return this.cache.GetOrAdd(headerName, r =>
    {
        var values = this.GetValue(r);
        var result = new List<Tuple<string, decimal>>();
        foreach (var header in values){
            {...}
            for (var index = 0; index < header.Length; index++){
                var character = header[index];
                if (character.Equals( ' ') && (index != header.Length -
1) && !isInQuotedSection){
                    continue;
                }
                if (character.Equals( '"')){
                    isInQuotedSection = !isInQuotedSection;
                }
                if (isInQuotedSection){
                    buffer += character;
                    if (index != header.Length -  1){
                        continue; ;
                    }
                }
                if (character.Equals( ';') || character.Equals( ',') ||
(index == header.Length -  1)){
                    if (!(character.Equals( ';') ||
character.Equals( ','))){
                        buffer += character;
                    }
                    if (isReadingQuality){
                        quality = buffer;
                    }
                    else{
                        if (name.Length > 0){
                            name +=  ';';
                        }
                        name += buffer;
                    }
                    {...}
                }
                if (character.Equals( ';')){
                    continue;
                }
                if ((character.Equals( 'q') || character.Equals( 'Q')) &&
(index != header.Length -  1)){
                    if (header[index + 1].Equals('=')){
                        isReadingQuality =  true;
                        continue;
                    }
                }
                if (isReadingQuality && character.Equals( '=')){
                    continue;
                }
                if (character.Equals( ',') || (index == header.Length -
1)){
                    var actualQuality = 1m;
                    decimal temp;
                    if (decimal.TryParse(quality, NumberStyles.Number,
CultureInfo.InvariantCulture,  out temp)){
                        actualQuality = temp;
                    }
                    {...}
                }
                buffer += character;
            }
        }
        return result.OrderByDescending(x => x.Item2);
    });
}
```

{...} Hidden code → Elements that impact cyclomatic complexity → New elements that impact cyclomatic complexity

Figure 4.8. Accessing method example (project Nancy).

4.4 Discussion e Implications

✓**Programming language plays an important role in the study of code complexity.** The evolution of complex methods is not necessarily equal among programming languages. For example, despite the majority of the complex methods increase complexity during evolution, C++ and Python ones present a higher proportion than other languages. In contrast, C# and Java ones present more effort to decrease complexity. *This way, to better understand code complexity evolution, researchers should not only focus their analysis on single programming languages. On the contrary: this study provides empirical evidence that each language has its distinction, thus, multi-language studies should be fomented.*

✓**Complex methods are not homogeneous in the operations they perform.** In addition to the variation of per language, we also find that complex methods vary according to the operation they perform (RQ3). That is, the complex methods themselves may have distinct goals and complexity trends, thus, being very heterogeneous. *Researchers should be aware of those distinctions among complex methods and strive to not treat them homogeneously to avoid loss of information and have more precise analysis.*

✓**Prioritize complex methods to be addressed during code cleaning.** We find that complex methods are over-concentrated on certain operations (*e.g., processing* and *conversion*) and under-concentrated on other operations (*e.g., accessing* and *test*). However, those operations are not equally problematic: *collection*, *conversion*, and *coordination* methods tend to become even more complex, whereas operations as *setup* and *release* barely increase complexity. *Those distinctions can be used to prioritize the complex methods to be addressed in code cleaning activities [46]. For example, in software systems with dozens of complex methods, which ones should be handled firstly by developers? In practice, this kind of prioritization could be integrated into smell detection tools (*e.g., *[17; 49; 54; 75]) so that developers can focus on the most critical operations beforehand.*

4.5 Threats To Validity

Method and file renaming. To ensure that we analyze the *entire* history of the complex methods, we manually examined the first commit of each method: when a method renaming was identified (either by name or by a change in its parameters) or moved from another file, it was discarded and a new method was evaluated (see Chapter 3).

That is, renamed/moved methods are not part of our dataset; in total, we discarded 735 methods. Another strategy to avoid loss of data is to ensure the full history of the file containing the method. Thus, we rely on the command `git -follow` to track files regardless of file renaming, ensuring that the entire file history is properly analyzed [2; 3]. Another solution we considered to identify the entire history is relying on the `git -L` command. However, we discarded this solution because it did not adequately identify the move method operations.

Wrongly migrated projects to Git/GitHub. In our study, the analysis of the entire project history is fundamental, thus, we ignored projects that for some reason do not make their full history available since the beginning (*e.g.,* they migrated from another platform to GitHub, are spin-off from another project, etc). To identify such repositories, we analyzed the very first commit of each candidate system and discarded the ones with more than 10 source files (see Chapter 3).

Minified JavaScript files and packages. JavaScript projects usually have copies of large files in minified formats [66]. This procedure is used to improve performance, decrease the size of source files, and promote more security in some scripts. However, the analysis of changes to these files would not be adequate because they have many instructions and functions in a few lines, making it impossible to identify which class or function was actually changed over time. Thus, we ignored files with these characteristics in JavaScript projects.

Classifying change types. In RQ2, we assessed the change types looking for keywords in commit messages. Thus, a set of keywords were used to identify bugs, new features, and refactoring changes [4; 35]. To validate this classification, we performed a manual evaluation of 384 randomly selected commits (*i.e.,* 95% confidence level and 5% confidence interval). This manual assessment of the classification using keywords led to a reasonable precision of 81%. Thus, the risk of wrong classifications in RQ2 is minimized.

Computing cyclomatic complexity. The computation of Cyclomatic Complexity was performed exclusively by the Lizard tool, which also provides data as NLOC and method location in the file (which is important to identify changes within the method). Thus, we are susceptible to possible internal tool errors during the analysis of over one million methods. Notice, however, that this tool is often adopted in the software industry, as mentioned in Chapter 3, which improves its reliability.

Generalization of the results. We analyzed 1,000 complex methods provided by 50 popular systems and written in five programming languages, making this study possibly

the most diverse in the literature so far. We recall that such type of study is often single language [9; 15; 20; 34; 41; 49; 51; 52; 53; 54; 56; 65; 72; 73; 74; 77; 79]. Moreover, our findings—as usual in empirical studies—may not be directly generalized to other systems, as commercial ones with closed source and implemented in other programming languages.

4.6 Final Remarks

In this chapter, we provided a quantitative study of the main characteristics of complex methods. We investigated three aspects of complex methods: (i) evolution over time; (ii) types of changes; and (iii) operations performed by the methods. The main findings for each question in this chapter are: (a) programming languages play an important role in monitoring method complexity; (b) complex methods are significantly over-represented on changes related to bugs, new features, and refactoring as compared to non-complex methods; and (c) complex methods implement different categories of operations, the largest concentration is *Processing*, *Conversion*, *Validation*, and *IO*.

The next chapter complements the quantitative analysis of this chapter, with a qualitative study. We answer RQ4 and RQ5, which seek to understand the developer's perception of complex methods.

Chapter 5

Survey Results: Developers' Perceptions on Complex Methods

In this chapter, we report the results of our survey with 73 developers, who have recently maintained complex methods in popular software systems. Our goal is to better understand the developers' perceptions of methods considered complex in multiple programming languages. Section 5.1 presents the result for RQ4, analyzing the first two survey questions. Then, Section 5.2 describes the main reasons for developers not to refactor complex methods, responding to RQ5. Section 5.3 describes the implications of the results. Lastly, we present the Threats to Validity in Section 5.4 and Final Remarks in Section 5.5.

5.1 RQ4: What are developers' perceptions of method complexity?

Question 1: *Do you think this complex method is harmful for maintenance?*

First of all, we identify whether the surveyed developers agree that the target method is complex or not. We find that that 49% of the developers (36 out of 73) do not consider the studied methods as complex, while 40% (29 out of 73) state that they are indeed complex. Also, the answer is not clear in 8 cases (11%). After identifying the developer's perceptions, we classify the rationales used in favor and against complexity, as follows.

Method is complex. Figure 5.1 presents the common reasons of the developers who classify the analyzed method as complex. Most of the developers agree that the target complex method ***compromises maintenance*** (53%, 18 answers). For

example, Developer #65 states: *"Yes, it's absolutely harmful for maintenance. Fixing existing bugs, implementing new features, addressing new requirements are all made much harder because of the complexity of this method"*. Similarly, Developer #43 says: *"Yes, actually the commit is to fix a bug which is somehow related to the complexity of this method, there are a few properties that need to be processed and it's very easy to cause a mistake like missing one property or typo from a copy-paste."*. Three developers (10%) state that the complex method ***discourages new contributors***. For example, Developer #34 mentions: *"As a new contributor, it took some effort to understand the working of the code and the data structures in play. This definitely raises the barrier of entry of contributors by quite a bit."*. Both categories agree that the complex methods are harmful to maintenance.

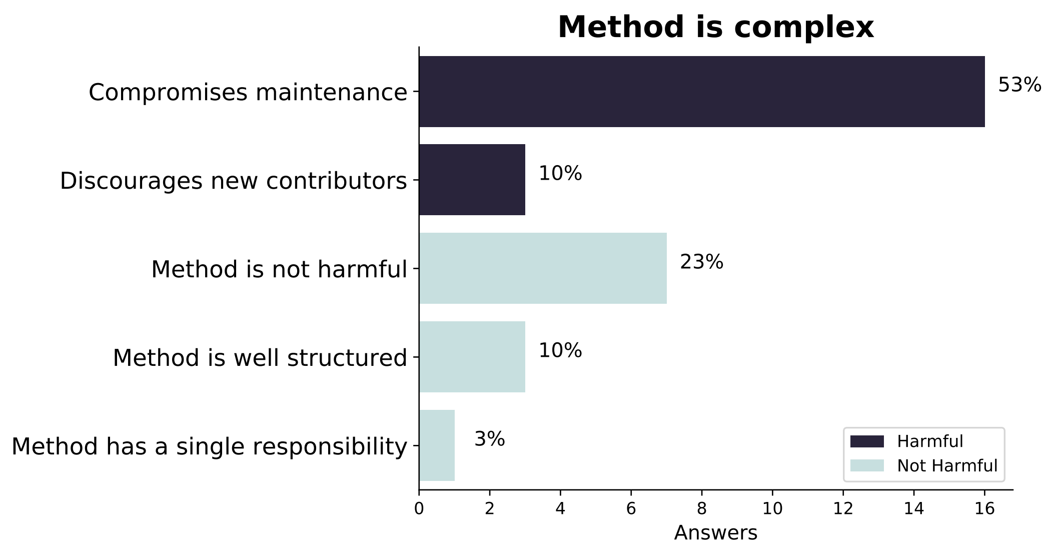


Figure 5.1. Developers who consider the methods as complex.

On the other hand, some developers considered the methods as complex, but stated that they are not necessarily harmful to maintenance (23%), they are well-written (10%), or they have single responsibility (3%). For example, Developer #39 writes: *"The original code mentioned in my PR is difficult to understand semantically. But I think it is not harmful (may by very very little ?) for maintenance."*. Similarly, Developer #46 mentions that the complex method is well-written and commented: *"Complex method can be harmful if what it does from a macro perspective isn't obvious and/or if it isn't rightfully commented. From the function name and the comment I understand what it does. No need to reverse engineer the code"*. Lastly, Developer #14 mentions the complex method is cohesive: *"Method has a single responsibility"*, thus, it is not hard to maintain.

Method is not complex. Figure 5.2 presents the common reasons of developers who do not consider the analyzed method as complex. The most commonly used arguments are ***cyclomatic complexity is a weak metric*** and ***method is well written*** (21% each). In this case, some developers criticized static metric analysis metrics, such as cyclomatic complexity. For example, Developer #13 states: *"I haven't ever found cyclomatic complexity to be a good measure for actual complexity in my career"*. It is interesting to note that we have not stated the metric we adopted, however, the developers seem to be aware that this metric is not always ideal. Other developers presented that the code was well written, thus, it would not have maintenance problems. For instance, Developer #25 mentions: *"The code here is actually well written and not complex in nature"*.

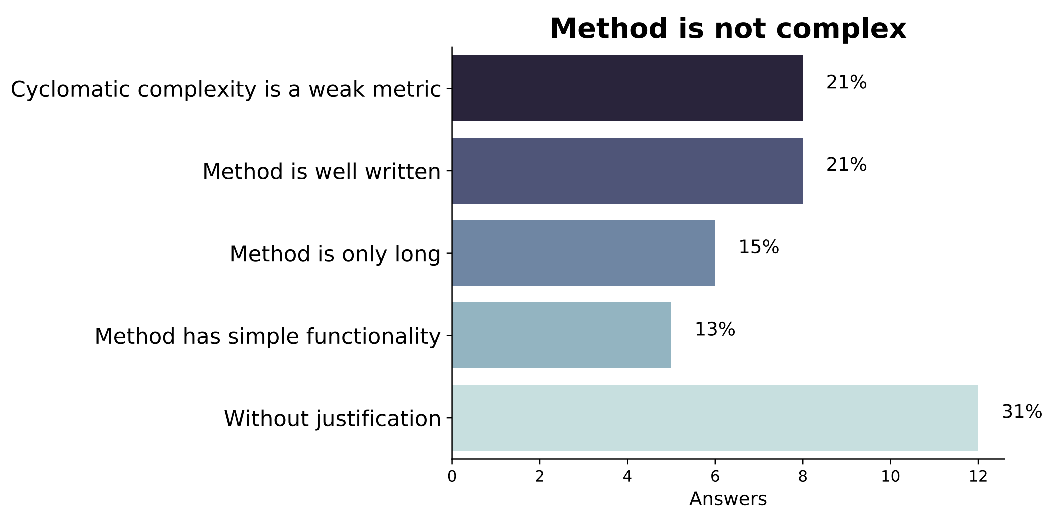


Figure 5.2. Developers who do not consider the methods as complex.

Next, we note that the categories ***method is only long*** (15%) and ***method has simple functionality*** (13%) are the other rationales against its complexity. In this context, Developer #06 states: *"This method is long, but not necessarily complex. It is a giant switch statement, and within each case, the logic is simple."*. Similarly, Developer #08 says: *"All it's really doing is extracting fields from a protobuf, validating their values, and storing them in a struct. I do not believe that this causes any maintenance burden."*. Lastly, other developers (31%) do not present clear rationales, for example, Developer #60 simply states: *"I don't think it's particularly complex"*.

Figure 5.3 summarizes the answers per programming language. Note that Java and C# have a higher proportion of developers who agreed that the target method is complex, with 71% and 67%, respectively. In contrast, in C++ and Python, only 37% and 35% of developers agree that the target method is complex, respectively. The

results support the results of our quantitative analysis, in the sense that there are differences between languages and that it is fundamental to approach each ecosystem differently.

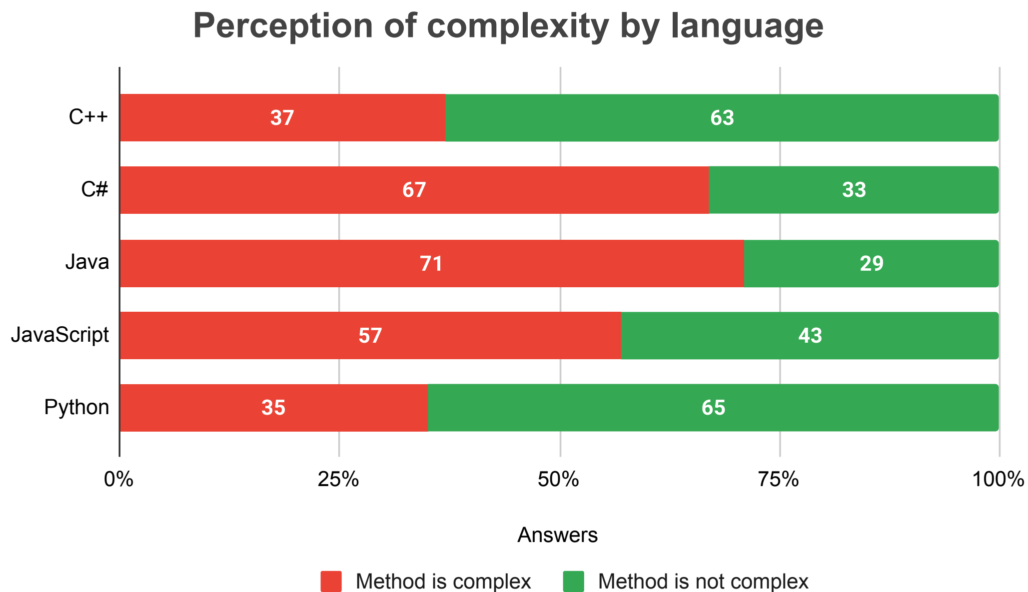
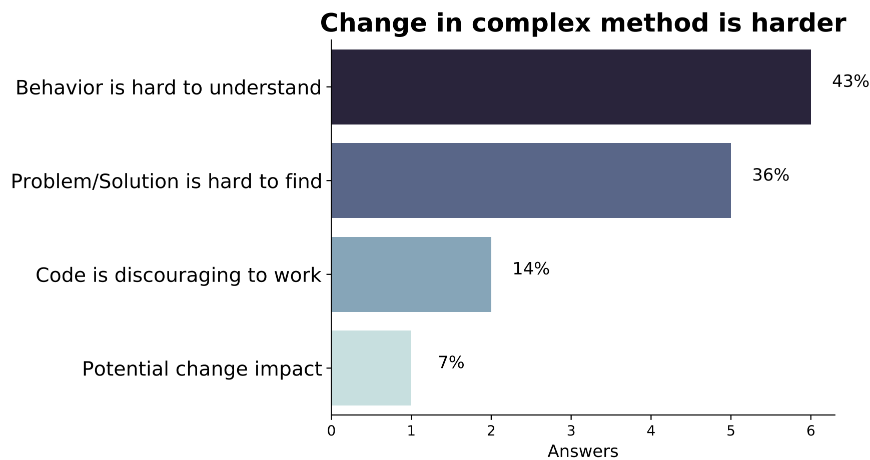


Figure 5.3. Proportion of answers per programming language.

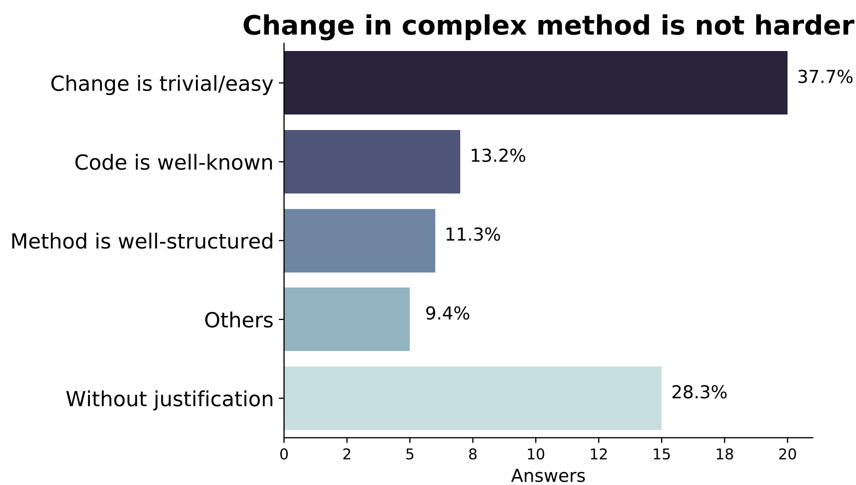
Question 2: *Was this change somehow harder to implement since it happened inside a complex method? Why?*

We summarize this analysis in two groups: developers who consider the change is hard (Figure 5.4a) and developers who consider the change is not hard (Figure 5.4b). **Change the complex method is harder.** The most used argument against modifying the target complex method is that its ***behavior is hard to understand***, with six responses (43%). Developer #65 states: *"Implementing the change in the commit I link above was much harder because of the complexity of the method in question. The change listed above took several weeks to implement, mostly because of needing to carefully understand the tangle of code."*. Similarly, Developer #40 mentions: *"yes, it was hard to track all the variables and what needed to altered during the refactor."*.

The second reason against ***problem/solution is hard to find*** (36%). In this case, the developers revealed problems to find bugs and propose solutions. Developer #33 states: *"Finding the exact cause and location of the bug was extremely difficult... And the size and complexity of the whole system, really not just this method, was an issue. In terms of localizing the bug, I was stepping through the debugger across multiple*



(a) Developers who consider the change is hard.



(b) Developers who not consider the change is not hard.

Figure 5.4. Developers' insights about how hard is to change complex methods.

functions, I didn't know at the outset that it was this function.". Likewise, Developer #19 mentions: *"The change itself is not hard to implement, but it's hard to find the problem.".*

Two developers revealed that the complex ***code is discouraging to work***. Finally, a single developer stated he had a problem with the modification due to ***potential change impact***. Developer #46 says: *"It was harder because of the pressure, the potential impact of a bug in such a core function. I force myself to write "perfect" code, because I absolutely didn't want to create new bug and also because I knew I would have to sell this changes to core maintainers."*

Change the complex method is not harder. In this group of answers, summarized in Figure 5.4(b), the main reason is ***change is trivial/easy*** (38%). Some developers explain that the change was easy because they do not need to understand or modify the method's logic. For example, Developer #24 says: *"No, because it was very specific and I don't care about the surrounding logic."*. Likewise, Developer #47 states: *"No, this changes was quite a primitive one, I've just used auxiliary method to prevent allocations when they are not necessary. The same change could be done automatically by static analysis tools."*.

In 13% of the cases, the developers stated they had no difficulty because the ***code is well-known***. In this context, Developer #02 says: *"No, possibly because I have worked with the codebase for some time now."*. In 11% (six answers), the developers said that the ***method is well-structured***, so this helped performing the change. For example, Developer #55 says: *"I found the rest of the method as well structured and easy to read. The code there is linear and every feature is separated"*.

Finally, the ***others*** category (9%) groups less frequent responses, such as method has automated tests (2 answers) and method was recently created (1 answer). Lastly, 15 developers (28%) just stated the change was not difficult, without any clear justification. Note that this category is different from *change is trivial/easy*, in which the developer explicitly claimed to have had great ease in implementing the changes. As an example, we cite the vague responses of Developers #20 and #60, respectively: *"This part of the change was not difficult to implement"* and *"The change wasn't hard to implement"*.

Summary of RQ4: We find no consensus regarding the developers' perceptions of method complexity. Methods with high cyclomatic complexity are considered complex by 40% of the developers and not complex by 50%:

- Developers in favor mostly say that the target complex method compromises maintenance and discourages new contributors. In contrast, developers against it state that cyclomatic complexity is a weak metric and that the target method is well-written or only long.

- Changing the target complex method is not necessarily harder because the change can be very specific and the developer does not need to know the surrounding logic. On the other hand, developers also state that the target complex method may have behavior that is hard to understand and that the problem/solution is hard to find.

5.2 RQ5: Why complex methods are not eliminated from code?

So far, it is clear that some developers may not consider the target methods as complex, as presented in RQ4. However, 40% of the respondent developers agree that the target methods are in fact complex. Thus, for these cases, one questions remain unanswered: why those complex methods were not refactored given they are known to be complex and, possibly, harmful for maintenance? This way, we proposed the following question in our survey:

Question 3: *Why do you think developers have never refactored such a complex method before?*

We find 10 reasons why the complex methods are not refactored, as summarized in Figure 5.5.

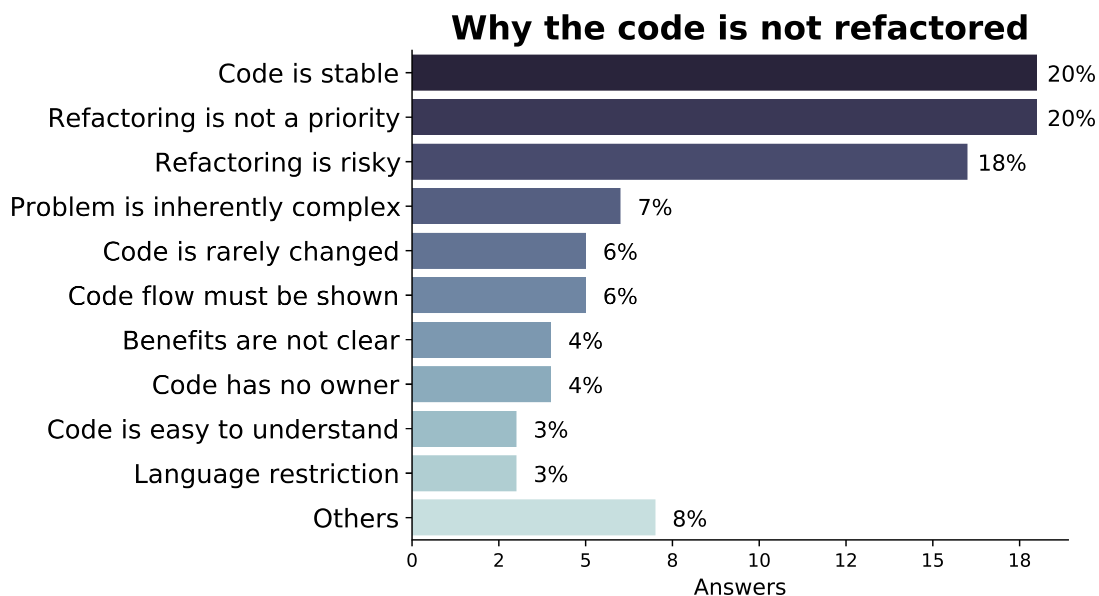


Figure 5.5. Reasons why complex methods are not refactored.

The two most frequent reasons to avoid refactoring are ***code is stable*** and ***refactoring is not a priority*** (20% each). For the first category, Developer #64 states: *"there wasn't a need as the function works fine, it has unit tests and there aren't new features requested"*. Similarly, Developer #08 says: *"The method has never been refactored because it's not necessary. The logic is perfectly understandable as-is."*, while Developer #20 mentions: *"My answer is that it is already well-structured to solve the problem. [...]"*. For the second category (*refactoring is not a priority*), other code mod-

ifications, such as feature addition bug-fixes, have a higher priority than refactoring. For example, Developer #43 states: *"I guess they have a lot of other tasks that have higher priority like bug fixing or supporting enterprise customers."*. Developer #04 states: *"Maybe because nobody cared and developers had other priorities? The former maintainers (HP, Google) abandoned the project, and there are only few volunteers now who do the work."*. Similarly, Developer #34 answers: *"When we are developing a feature, we sometimes focus on getting it work first and then make it work right. Sometimes, some refactorings might missed, or maybe not worth the effort"*.

Refactoring is risky is another frequent reason (18%). In this case, the developers state that code modification may add bugs and break other parts of the system. For example, Developer #21 mentions: *"From my experience the most common reasons I have seen are: 1. Bad risk/reward ratio. Refactoring code is a risky job. 2. Implicit dependencies on the implementation. Risk of breaking app compatibility. [...]"*. Similarly, Developer #63 says: *"Refactoring this particular code structure would require refactoring ALL of the affected modules, which in an open source project of that scale becomes far more complex and dangerous than leaving it as is."*. Indeed, to avoid this scenario, refactoring should always be performed with tests, ensuring that the behavior of the code is not changed [22].

Next, we have the category ***problem is inherently complex*** (7%). This states that complexity can not be avoided due to the complex nature of the addressed problem. Developer #01 mentions: *"There is a base complexity in the functionality that this method provides. This base complexity cannot be avoided. [...]"*. Developer #13 comments: *"It was refactored many times, but there's no good way to implement something beautifully that's inherently ugly"*.

Code flow must be shown is reported by 6% of the developers. In this case, they claim that it is necessary to visualize the code flow in a single method or screen, for example, without browsing the code. Developer #42 mentions: *"I think it was not refactored because there is always a point where you should expose complexity. This method does so in a quite clean way. [...]"*. Developer #26 says: *"Sometimes splitting something into multiple functions is better, but sometimes it isn't. Functions which perform a long sequence of operations which cannot easily be shared with other functions are often easier to understand as single functions than they would be with extra abstraction"*. Likewise, Developer #58 states: *"The only thing you can do is to split off reading the file itself to a separate function. That may make testing some thing easier. On the other hand, it may be harder to follow the complete flow of the data."*

The category ***code is rarely changed*** has 6% of the responses. Developer #05 mentions: *"It's isolated, rarely touched [...]"*. Similarly, Developer #33 says: *"[...] it*

also is not a heavily modified section of the code relatively speaking, I think.". There are also less frequent categories: ***benefits are not clear***, ***code has no owner***, ***code is easy to understand***, and ***language restriction***, with 4, 4, 3, and 3 answers, respectively. Finally, we grouped some less adopted reasons in the ***others*** category, including issues related to performance, optimization, and code duplication.

Summary of RQ5: Complex methods are not refactored mainly due to three reasons: (1) developers consider the methods stable in their current state way; (2) refactoring is not considered a high-priority task, like bug-fixing; and (3) refactoring is considered risky, possibly, due to lack of tests. Other reasons include: the problem is inherently complex, code is rarely changed, and code flow must be shown.

5.3 Discussion and Implications

✓**The developers' perception of complexity is subjective and varies programming language.** In our survey, we find no consensus regarding the developers' perceptions of method complexity. Methods with high cyclomatic complexity are considered complex by 40% of the developers and not complex by 50% (RQ4). Moreover, it is interesting to note that the programming languages with the lowest perception of complexity are the ones with the highest trends in increasing complexity (RQ1), and vice-versa. For example, by contrasting Figures 4.2 and 5.3, we observe that C++ and Python have the highest rate of methods increasing complexity over time, while they have the lowest perception of complexity. On the other hand, Java and C# have the highest rate of methods decreasing complexity over time, while they have the highest perception of complexity. *Thus, the perception of complexity on each programming language may play a role in the way methods are developed and efforts are allocated to reduce complexity. Further research should be performed per language to deep understand this issue.*

✓**Developers may deliberately avoid refactoring complex code.** We find that developers do not refactor complex methods due to several reasons (RQ5), including *code is stable*, *problem is inherently complex*, and *code flow must be shown*, to name a few. That is, sometimes, developers are satisfied with complexity or even want to expose it. Moreover, 3 out of the 10 reasons reported by developers directly blame the refactoring activity itself: *refactoring is not a priority*, *refactoring is risky*, and *benefits are not clear*. *We thus shed light on the contrast between theory and practice: despite the clear benefits of refactoring to improve maintainability [22; 46], in practice, developers may deliberately avoid refactoring complex code due to distinct reasons.*

5.4 Threats To Validity

Survey study. The survey analysis has been performed with special attention by the two authors via *thematic analysis* to minimize subjectivity. The reliability of responses can also be considered a threat to the survey study. Thus, to reduce this risk, we assessed the most recent commits to finding recent changes that were likely to be fresher in the memory of the developers. Moreover, most respondent developers come from large software companies or work in worldwide projects hosted in GitHub. Finally, we assessed the level of commits of the respondent developers as compared with average ones, and we find that most respondents (64%) have high activity levels (see Chapter 3).

Generalization of the results. We focused our survey on developers who recently modified complex methods in popular GitHub repositories, involving five programming languages (*i.e.,* C#, C++, Python, Java, and JavaScript). Then, we interpreted and categorized responses from 73 developers to the three questions addressed. Therefore, as mentioned in threats in Chapter 4 and empirical software engineering, our findings are restricted to the study's subject and cannot be generalized to other scenarios.

5.5 Final Remarks

In this chapter, we presented the second study of this dissertation: a survey with contributors of popular GitHub projects. We investigated three questions to understand developers' perceptions of source code complexity. Specifically, we performed a qualitative study to investigate: (i) the impact of complexity on maintenance tasks; (ii) the difficulty to maintain complex methods; and (iii) the reasons for the existence of complex methods. As result, we identified that there is no consensus on the developers' perception of complexity. As such, certain contributors have indicated that complexity compromises maintenance and discourages new contributors. In contrast, others have stated that cyclomatic complexity is a weak metric for this type of analysis. Similar behavior is noted when evaluating changes applied by developers. Finally, as the main reasons for not refactoring complex methods, we can mention: *Code is stable*, *Refactoring is not a priority*, and *Refactoring is risky.*

In the next chapter, due to the lack of consensus on complexity, we deepen our analysis of complex methods, looking for evidence that differentiates the methods considered complex and not complex by the developers.

Chapter 6

Assessing Self-Admitted Complex Methods

This chapter presents the third and final study of the dissertation. This analysis answers the last research question (RQ6) and assesses *self-admitted complex methods*, that is, methods that the developers themselves classify as complex. Section 6.1 explores the three studied aspects: Developer Experience, Code and Evolution, and Maintenance. Next, in Section 6.2, we present and discuss the findings and implications. Finally, Sections 6.3 and 6.4 present the Threats to Validity and Final Remarks, respectively.

6.1 RQ6: To what extent are self-admitted complex methods different from other complex methods?

In this analysis, we considered 65 of the 73 survey developers' answers, ignoring eight responses due to the impossibility of identifying the developer's real perception of the method's complexity. Thus, we removed four developers from the C++ programming language (*i.e.,* two from project rocksdb, one from folly, and one from v8), three Python developers (*i.e.,* two from cpython and one from ansible), and one JavaScript developer from the angular.js repository.

As mentioned at the beginning of Section 5.1, considering now only 65 developers, we find that 55% (36) of the developers denied that the target method was complex; we call those methods *self-admitted not complex methods.* On the other hand, 45% (29) agree that the method was complex; those methods are called *self-admitted complex*

methods. Below we discuss the answers to the three aspects addressed in this research question.

6.1.1 Developer Experience

According to the NOC (*i.e.*, number of commits), Figure 6.1 summarizes this first analysis: it breaks the experience of the surveyed developers according to their perceptions of complexity. We first notice that most of the surveyed developers have high experience (red bar) as compared to the other developers in the same project. Interestingly, the most experienced developers are more likely to classify the target methods as not complex (69%) than as complex (55%). On the other hand, developers with less experience (blue bar) are more likely to classify the target methods as complex (24%) than as not complex (8%).

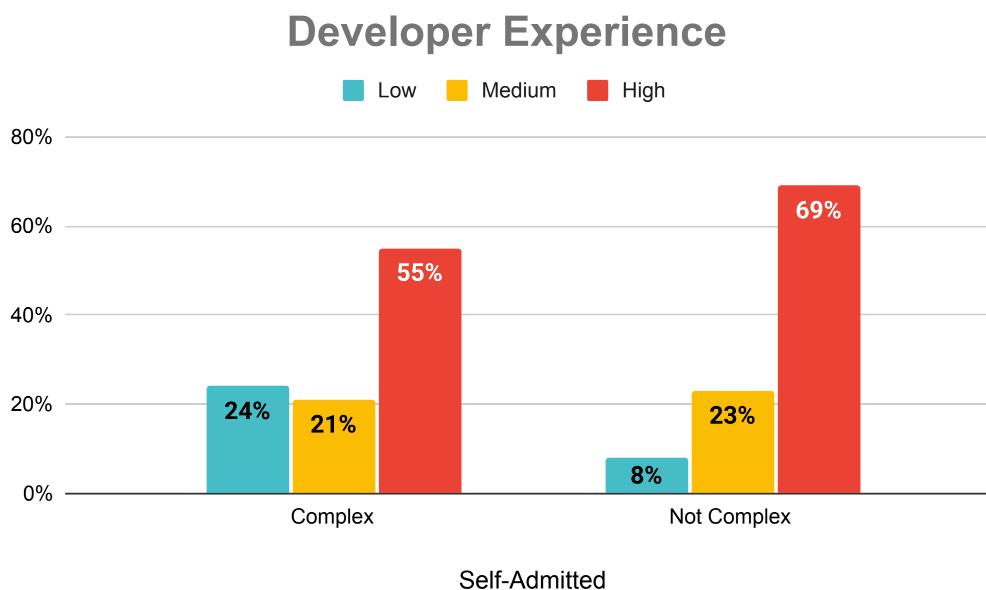


Figure 6.1. Experience of the surveyed developers vs. perceptions of complexity.

Summary of RQ6a: The majority of the surveyed developers have a high level of experience in the analyzed projects. *The experience of the developers may play a role in their perceptions of code complexity.* That is, the most experienced developers are more likely to classify the target methods as *not complex*, while the least experienced developers are more likely to classify the target methods as *complex*.

6.1.2 Code and Evolution

Next, we aim to explore the code and evolutionary differences between the two groups of complex methods. As presented in Figure 6.2, on the median, both groups of methods have a complexity of 15 (the difference is not statistically significant for the Mann-Whitney test, *p-value*=0.25). In other words, both self-admitted complex and not complex methods are equally complex.

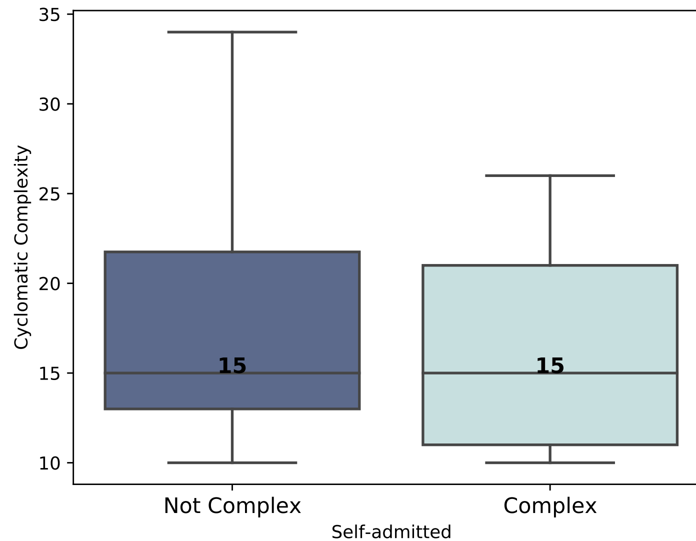


Figure 6.2. Complexity of the analyzed methods.

Table 6.1 explores other five metrics: lines of code, tokens (*i.e.,* number of words and operators), number of parameters, number of commits that changed the target method, and growth rate (*i.e.,* the current complexity of the method divided by the complexity of its very first version). On the median, the self-admitted complex methods ("SA C") have 78 lines of code, while self-admitted not complex methods ("SA NC") have 74. Both groups also have an equivalent number of parameters (*i.e.,* 2 parameters, on the median) and commits that changed the target method (around 6 changing commits, on the median). Column *p-value* presents the results of the comparison between the groups for the Mann-Whitney test. As we notice, none of the comparisons is statistically significant (*i.e.,* all *p-values* ≥ 0.05), meaning that the groups are equivalent regarding the analyzed metrics.

Summary of RQ6b: Self-admitted complex methods are equivalent to other complex methods in terms of complexity, lines of code, tokens, parameters, commits, and growth rate. In other words, those factors cannot explain their differences.

Table 6.1. Code and evolution of the complex methods (median values).

Dimension	Metric	SA NC	SA C	*p-value*
Code	Lines of code	74	78	0.48
	Tokens	425.5	332	0.35
	Parameters	2	2	0.36
Evolution	Commits	6.5	6	0.27
	Growth rate	131%	105%	0.43

6.1.3 Maintenance

Lastly, we seek to understand how both groups of complex methods affect software maintenance. As a proxy of maintainability, we analyze the issues and pull requests (PRs) related to each complex method. We find 32 issues and pull requests referencing the self-admitted complex methods and 38 issues and pull requests about the other complex methods. For comparison purposes, we randomly select 70 ordinary issues and pull requests, following the proportion of programming languages of the interviewed developers. Based on that data, we extract two metrics: (a) number of messages in issues and pull requests and (b) number of changed files in pull requests. The rationale is that the higher those numbers, the more discussion and effort are dedicated to the proposed changes.

Figure 6.3a shows that issues and pull requests of the self-admitted complex methods have 10 messages, while the other complex methods have 6.5 messages (for comparison, ordinary issues/pull requests have only 3 messages). Notice that their pull requests have 3 and 2.5 changed files, respectively (Figure 6.3b). Despite the apparent distinction, we find no statistically significant difference between both groups of complex methods (*i.e.,* all *p-values* ≥ 0.05), meaning they are equivalent. However, when comparing the self-admitted complex methods with the ordinary ones, the difference is significant (with moderate and very large effect for the Cohen effect size).

To further explore possible maintainability issues in the target methods, we assess the most common maintenance problems on their issues and pull requests. For this purpose, we rely on thematic analysis [14] to identify and record themes. Specifically, we perform the following steps: (1) initial reading of the issues/PRs, (2) generating a first code for each issues/PRs, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes.

As summarized in Table 6.2, we find seven major maintenance problems in the investigated methods: lack of tests, hard maintenance, presence of bugs, low perfor-

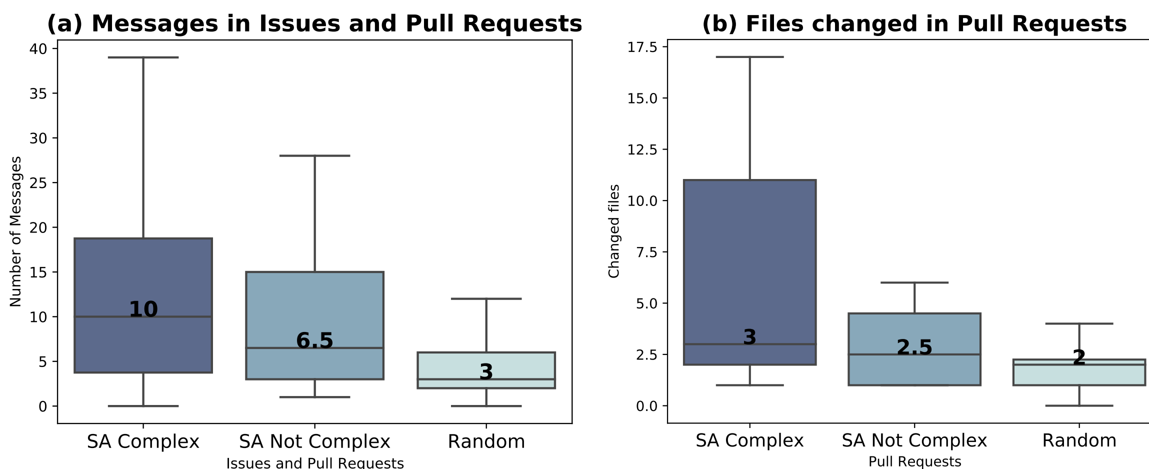


Figure 6.3. Distribution of (a) messages and (b) changed files in issues/PRs.

mance, lack of documentation, bad nomenclature, and code duplication. The columns "SA NC" (self-admitted not complex methods) and "SA C" (self-admitted complex methods) detail the frequency of each maintenance problem.

Overall, we could not find any major difference between the two groups of complex methods, despite the self-admitted complex methods have slightly more maintenance problems than the other complex methods (26 vs. 21). Thus, we can roughly state that both groups of complex methods suffer from the same types of maintenance problems. Also, we recognize that this dataset is small and should be expanded by further studies for more robust analysis.

Summary of RQ6c: We did not find significant differences when we evaluated the issues and pull requests of the target methods. In contrast, we identified the following similarities according to issues and pull requests:

- Self-admitted complex methods and other complex methods have an equivalent number of messages and changed files, suggesting the same level of discussion and effort.
- Self-admitted complex methods and other complex methods have similar maintenance problems, such as lack of tests, presence of bugs, and low performance.

Table 6.2. Maintenance problems found in issues and pull requests of complex methods.

Problem	Example	#SA NC	#SA C
Lack of Tests	*"Could you add a test case to verify that assigning unsaved objects to OneToOneField and GenericForeignKey aren't also affected by this issue?"*	8	4
Hard Maintenance	*"random is the major outlier here, as it has a not-quite-trivial implementation for Windows, and a pretty complex implementation for non-Apple, non-Windows targets"*	6	10
Presence of Bugs	*"Fixing those two things leads to crashes elsewhere from the little that I tested, so there's probably a third problem somewhere"*	3	3
Low Performance	*"I myself have experienced the slow cursor movement... It is called many times during screen redrawing. All those vector initializing and copying consumes lots of CPU cycles"*	2	1
Lack of Docs.	*"Just reading this docsrting it's unclear what function it could be"*	1	3
Bad Nomenclature	*"Their names are also changed to add "_swift" to the front to match our naming conventions"*	1	2
Code Duplication	*"We could try to reduce the duplicated logic by factoring it out into an InsertExplicitCall::attempt method"*	0	3
All	-	21	26

6.2 Discussion e Implications

✓**The experience of the developers may affect their perceptions of complexity.** In our analysis, we find that the developer's experience impacted the perceived complexity of the methods. About three times more developers with low experience found the methods complex. Concomitantly, more developers with more experience considered the methods as non-complex (RQ6a). *In this way, complexity can be a problem for novice developers, as we also report in the RQ4 responses (*i.e., *Discourages new contributors). Thus, we must consider the developer's experience when analyzing*

complexity in source code.

✓**Differentiating self-admitted complex methods from other complex methods is not a trivial task.** Our main objective in this RQ6 was to seek characteristics that differentiate self-admitted complex methods from self-admitted not complex methods, but our findings demonstrated that these two groups are more similar than distinct. Thus, the methods are considered similar both in terms of code and evolution (RQ6b) and in the problems identified in the issues and pull requests related to the target methods (RQ6c). *Since current code metrics are not sufficient to assess code complexity as perceived by developers, researchers should be aware of the need for further studies to automatically identify self-admitted complex methods.*

6.3 Threats To Validity

Manual classification of the issues and pull requests. The classification of the categories in RQ3 was performed manually by the authors of the paper. To reduce the subjectiveness of this analysis, we rely on thematic analysis [14].

Generalization. We analyzed dozens of complex methods provided by popular open-source systems, which are written in five programming languages (JavaScript, Python, Java, C++, and C#). Despite these observations, our findings—as usual in empirical studies—may not be directly generalized to closed-source systems and other programming languages.

6.4 Final Remarks

In this chapter, we presented the final study of this master dissertation. We investigated self-admitted complex methods according to the developers' perceptions. By evaluating 65 target methods in three aspects (*i.e.,* Developer Experience, Code and Evolution, and Maintenance), we obtained the following results: (i) the experience of the developers may play a role in their perceptions of code complexity; (ii) self-admitted complex methods are similar to other methods in terms of code and evolution; and (iii) self-admitted complex methods have the same characteristics and problems in the discussion of issues and pull requests as other methods.

In the next chapter, we present our conclusions, summarizing the results of our analysis and discussing the main contributions of this dissertation. In addition, we propose future research directions.

Chapter 7

Conclusion

7.1 Summary and Contributions

In this dissertation, we provided a multi-language empirical study to assess the evolution of complex methods, a survey study to better understand developers' perceptions, and a comparative study on self-admitted complex methods and other methods. We analyzed 1,000 complex methods of 50 popular projects written by five programming languages (JavaScript, Python, Java, C++, and C#) and we perform a survey with over 70 developers. Overall, we found that programming language plays an important role in the study of code complexity and that complex methods are not homogeneous in the operations they perform. The developers' perception of complexity is subjective and varies per programming language. Furthermore, developers may deliberately avoid refactoring complex code due to several reasons, including *code stability*, *lack of refactoring priority*, and *refactoring risk*. Finally, according to the metrics covered, we verified that self-admitted complex methods are not easily distinguishable from other complex methods. In contrast, we found that the developer's experience may have influenced their perception of complexity.

Based on our results, we provided insights to both researchers and practitioners. We presented that the studied programming languages have close but not equal issues regarding method complexity, thus, researchers should not focus their analysis on single languages. Indeed, we showed that, independently of the programming language, complex methods are living entities that tend to change frequently and grow in complexity. Our survey results presented that the perception of complexity is subjective and varies per programming language. Then, we discussed the reasons why complex methods are deliberately not refactored over time. Finally, in our last and preliminary study, we could not find any major difference between self-admitted complex methods

and other complex methods, analyzing aspects of code, evolution, and maintenance.

7.2 Future Work

We consider that our work can be complemented with the following future work:

Evaluate tests of complex methods. As future work, we would like to analyze whether complex methods are covered by tests and what are the characteristics of these tests. In this way, it would be possible to identify the types of tests. Whether unitary, integrating, or functional. Also, we would compare the complexity of tests that evaluate complex methods with other tests, aiming to identify whether complex methods generate "complex tests".

Investigate the evolution of other code smells. We plan to investigate other code smells and anti-patterns available in the literature, such as code duplication and long parameter lists. In this way, we can identify the evolution of these elements over time, comparing the evolution of the complexity of the methods.

Improve complexity metrics. Since cyclomatic complexity is not a very precise metric to identify complex methods, there is a need to improve it so that more practitioners and researchers can adopt it. For best results, other metrics involving method naming, depth, and code duplication can be incorporated or grouped at cyclomatic complexity.

Explore other characteristics of self-admitted complex methods. In our study, we shed light on common factors between the two groups of complex methods. This opens room for novel researches to explore other factors to capture the developers' perception of complexity, for example, security, comprehensibility, and readability, to name a few.

Bibliography

[1] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen. Code smells for Model-View-Controller architectures. *Empirical Software Engineering*, 23:2121–2147, 2018.

[2] G. Avelino, L. Passos, A. Hora, and M. T. Valente. A novel approach for estimating truck factors. In *24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[3] G. Avelino, L. Passos, A. Hora, and M. T. Valente. Measuring and analyzing code authorship in 1+118 open source projects. *Science of Computer Programming*, 176(1):14–32, 2019.

[4] Y. Ayalew and K. Mguniin. An assessment of changeability of open source software. *Computer and Information Science*, 6(3):68–79, 2013.

[5] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of GitHub repositories. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

[6] M. W. Bray, K. Brune, D. Fisher, J. Foreman, and M. Gerken. C4 software technology reference guide - a prototype. 1997.

[7] A. Brito, M. T. Valente, L. Xavier, and A. Hora. You broke my code: understanding the motivations for breaking changes in apis. *Empirical Software Engineering*, 25:1458 – 1492, 2020.

[8] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of bad smells in object-oriented code. In *International Conference on the Quality of Information and Communications Technology*, pages 106–115. IEEE, 2010.

[9] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering*, 10(1):3–18, 2014.

[10] Z. Chen, L. Chen, W. Ma, and B. Xu. Detecting code smells in python programs. In *International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 18–23, 2016.

[11] Chi-Square - College of Education - NIU, October, 2020.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[13] D. Cruz, A. Santana, and E. Figueiredo. Detecting bad smells with machine learning algorithms: An empirical study. In *International Conference on Technical Debt*, page 31–40, 2020.

[14] D. S. Cruzes and T. Dyba. Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275–284, 2011.

[15] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.

[16] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 341–350, 2015.

[17] A. M. Fard and A. Mesbah. JSNOSE: Detecting JavaScript Code Smells. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013.

[18] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. A review-based comparative study of bad smell detection tools. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–12, 2016.

[19] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance (ICSM)*, pages 23–32, 2003.

[20] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5–1, 2012.

[21] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21:1143–1191, 2016.

[22] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[23] F. Grund, S. A. Chowdhury, N. Bradley, B. Hall, and R. Holmes. Codeshovel: Constructing method-level source code histories. 2021.

[24] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *International Conference on Software Engineering (ICSE)*, pages 392–401, 2013.

[25] M. Hilton, J. Bell, and D. Marinov. A large-scale study of test coverage evolution. In *ACM/IEEE International Conference on Automated Software Engineering*, pages 53–63, 2018.

[26] A. Hora, N. Anquetil, S. Ducasse, and S. Allier. Domain specific warnings: Are they any better? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 441–450, 2012.

[27] A. Hora and R. Robbes. Characteristics of method extractions in java: A large scale empirical study. *Empirical Software Engineering*, 25:1798–1833, 2020.

[28] A. Hora, D. Silva, R. Robbes, and M. T. Valente. Assessing the Threat of Untracked Changes in Software Evolution. In *International Conference on Software Engineering (ICSE)*, pages 1102–1113, 2018.

[29] E. W. Host and B. M. Ostvold. The programmer's lexicon, volume i: The verbs. In *International Working Conference on Source Code Analysis and Manipulation*, pages 193–202, 2007.

[30] A. Jbara, A. Matan, and D. G. Feitelson. High-mcc functions in the linux kernel. *Empirical Software Engineering*, 19:1261–1298, 2014.

[31] D. Johannes, F. Khomh, and G. Antoniol. A large-scale empirical study of code smells in JavaScript projects. *Software Quality*, 27:1271–1314, 2019.

[32] M. G. Kendall. Rank correlation methods. *Griffin*, 1948.

[33] F. Khomh, M. Di Penta, and Y. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Working Conference on Reverse Engineering*, pages 75–84, 2009.

[34] F. Khomh, S. Vaucher, Y. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *International Conference on Quality Software*, pages 305–314, 2009.

[35] S. Kim, E. J. Whitehead,, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[36] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. *International Workshop on Principles of Software Evolution (IWPSE)*, 09 2001.

[37] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, 2007.

[38] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[39] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124. Springer, 1996.

[40] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution-the nineties view. In *International Software Metrics Symposium*, pages 20–32. IEEE, 1997.

[41] H. Liu, X. Gong, L. Liao, and B. Li. Evaluate how cyclomatic complexity changes in the context of software evolution. In *Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 756–761, 2018.

[42] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *European Conference on Software Maintenance and Reengineering*, pages 277–286, 2012.

[43] E. d. S. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[44] N. Maneerat and P. Muenchaisri. Bad-smell prediction from software design model using machine learning techniques. In *International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 331–336, 2011.

[45] H. B. Mann. Nonparametric tests against trend. *Econometrica*, 13(3):245–259, 1945.

[46] R. C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[47] T. McCabe and A. Watson. Software complexity. *Crosstalk: The Journal of Defense Software Engineering*, 7(12):5–9, December, 1994.

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

[49] N. Moha, Y. Gueheneuc, L. Duchien, and A. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[50] W. Oizumi, A. Garcia, L. Da Silva Sousa, B. Cafeo, and Y. Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *International Conference on Software Engineering (ICSE)*, pages 440–451, 2016.

[51] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *International Symposium on Empirical Software Engineering and Measurement*, pages 390–400. IEEE, 2009.

[52] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[53] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *International Conference on Software Maintenance and Evolution*, pages 101–110. IEEE, 2014.

[54] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *International Conference on Automated Software Engineering (ASE)*, pages 268–278, 2013.

[55] F. Pecorelli, F. Palomba, and A. De Lucia. The relation of test-related factors to software quality: A case study on apache systems. *Empirical Software Engineering*, 26(2):1–42, 2021.

[56] R. Peters and A. Zaidman. Evaluating the lifespan of code smells using software repository mining. In *European Conference on Software Maintenance and Reengineering*, pages 411–416, 2012.

[57] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 91–100, 2014.

[58] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, page 35–38, 2008.

[59] G. Robles, I. Herraiz, D. M. German, and D. Izquierdo-Cortazar. Modification and developer metrics at the function level: Metrics for the study of the evolution of a software project. In *International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 49–55, 2012.

[60] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol. An empirical study of code smells in JavaScript projects. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305, 2017.

[61] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 1(1):1–17, 2020.

[62] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of GitHub contributors. In *International Symposium on the Foundations of Software Engineering*, pages 858–870, 2016.

[63] D. Silva and M. T. Valente. RefDiff: detecting refactorings in version histories. In *International Conference on Mining Software Repositories*, pages 269–279, 2017.

[64] H. Silva and M. T. Valente. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

[65] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[66] P. Skolka, C.-A. Staicu, and M. Pradel. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *The World Wide Web Conference*, page 1735–1746, 2019.

[67] E. V. Sobrinho, A. De Lucia, and M. Maia. A systematic literature review on bad smells–5 w's: Which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66, 2021.

[68] D. Spadini, M. Aniche, and A. Bacchelli. Pydriller: Python framework for mining software repositories. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, 2018.

[69] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, pages 10–10, 2007.

[70] D. Taibi, A. Janes, and V. Lenarduzzi. How developers perceive smells in source code: A replicated study. *Information and Software Technology*, 92:223 – 235, 2017.

[71] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *International Conference on Software Engineering*, pages 483–494, 2018.

[72] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *International Conference on Software Engineering*, volume 1, pages 403–414. IEEE, 2015.

[73] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.

[74] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106, 2002.

[75] N. Vavrová and V. Zaytsev. Does Python Smell Like Java? Tool Support for Design Defect Discovery in Python. *Computing Research Repository*, abs/1703.10882, 2017.

[76] L. Xavier, F. Ferreira, R. Brito, and M. T. Valente. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In *International Conference on Mining Software Repositories*, page 137–146, 2020.

[77] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *International Conference on Software Maintenance (ICSM)*, pages 306–315, 2012.

[78] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *Working Conference on Reverse Engineering (WCRE)*, pages 242–251. IEEE, 2013.

[79] A. Yamashita and L. Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *International Conference on Software Engineering (ICSE)*, pages 682–691, 2013.

[80] S. Yue, P. Pilon, and G. Cavadias. Power of the mann–kendall and spearman's rho tests for detecting monotonic trends in hydrological series. *Journal of Hydrology*, 259(1):254 – 271, 2002.

[81] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9, 2007.