



Run Run Shaw Library

香港城市大學
City University of Hong Kong

Copyright Warning

Use of this thesis/dissertation/project is for the purpose of private study or scholarly research only. ***Users must comply with the Copyright Ordinance.***

Anyone who consults this thesis/dissertation/project is understood to recognise that its copyright rests with its author and that no part of it may be reproduced without the author's prior written consent.

CITY UNIVERSITY OF HONG KONG
香港城市大學

Improving Code Readability Classification
Using Deep Learning Techniques
基於深度學習技術的代碼可讀性分類研究

Submitted to
Department of Computer Science
電腦科學系
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
哲學博士學位

by

Mi Qing
米慶

August 2018
二零一八年八月

Abstract

Code readability classification (which refers to classification of a piece of source code as either readable or unreadable) has attracted increasing concern in academia and industry. To build accurate classification models, existing studies mainly consist of two phases: 1) Collect labeled data by conducting a large-scale survey, inviting multiple human annotators (preferably domain experts) to rate code snippets by readability; 2) Handcraft features from different aspects that intuitively seem to have some effect on code readability, and then train a machine learning classifier using data collected from the first phase.

Despite the encouraging results, several problems have been identified with existing approaches. For instance, in Phase 1, the process of code readability survey has long been criticized as laborious and tedious (Problem 1), while in Phase 2, the widely used feature engineering method is labor-intensive and can capture only partial information about the source code (Problem 2). In this thesis, we deal with these two problems by using several latest techniques.

To address Problem 1, we introduce a new interactive method with the advent of gamification techniques and implement it as GamiCRS (A Gamification System for Code Readability Survey). The focus is on incorporating game-based mechanisms to enable participants with positive attitudes towards a more interesting survey process, with the underlying goal of improving the quality of survey results. A complete incentive and reward model

is proposed together with a combination of both intrinsic and extrinsic motivators identified. To ensure its dynamic efficacy, a field experiment is carried out to compare GamiCRS with its non-gamified counterparts. The empirical results show a positive effect towards the application of GamiCRS.

To address Problem 2, we introduce a deep learning-based approach that can learn complicated underlying features automatically from the source code. Our approach is not another model based on feature engineering, but a constructive method of gaining a deep understanding of what constitutes a readable code, which makes our study novel in this regard. Specifically, we treat a source code as a matrix of symbols, and leverage Convolutional Neural Networks (ConvNets) to automatically learn features directly from the input data. Our approach has the advantage of requiring no human intervention and thus can effectively avoid personal biases and neglects of certain features.

First, we formulate a strategy for source code representation to enable deep learning-based program analyses. In particular, we parse the source codes into a set of symbols according to different granularities (i.e., *Character-Level*, *Token-Level*, and *Node-Level Representation*), and then convert the symbols into integer matrices, which is the universal format that ConvNets can easily analyze and manipulate. Corresponding to each granularity, we construct three separate ConvNets with identical architectures. The objective is to have multiple ConvNets that are skillful, but from different perspectives. In order to improve model generality and applicability, we then aggregate the three ConvNets with adjustable weights. We denote the resulting model as DeepCRM (A Deep Learning-Based Code Readability Model).

To evaluate whether deep learning is beneficial, we compare our approach with five state-of-the-art code readability models, namely, *Buse et al.'s Model*, *Posnett et al.'s Model*,

Dorn et al.'s Model, *Scalabrino et al.'s Model*, and *A Comprehensive Model*. The experimental results show that DeepCRM outperforms previous approaches. The improvement in accuracy ranges from 2.4% to 17.2%, confirming the efficacy of deep learning techniques in the task of code readability classification. Although deep learning has achieved remarkable success in other areas, to the best of our knowledge, we are the first to observe its effectiveness in code readability classification. We hope that the promising results will interest and encourage more in-depth research in this new field.

CITY UNIVERSITY OF HONG KONG
Qualifying Panel and Examination Panel

Surname: MI
First Name: Qing
Degree: PhD
College/Department: Department of Computer Science

The Qualifying Panel of the above student is composed of:

Supervisor(s)

Dr. KEUNG Wai Jacky Department of Computer Science
City University of Hong Kong

Qualifying Panel Member(s)

Dr. CHAN Wing Kwong Department of Computer Science
City University of Hong Kong

Dr. LI Zhenjiang Department of Computer Science
City University of Hong Kong

This thesis has been examined and approved by the following examiners:

Dr. KWOK Lam For Department of Computer Science
City University of Hong Kong

Dr. LI Minming Department of Computer Science
City University of Hong Kong

Dr. KEUNG Wai Jacky Department of Computer Science
City University of Hong Kong

Dr. LAI Richard Y Department of Computer Science and Information
Technology
La Trobe University

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Dr. KEUNG Wai Jacky, for his great effort and continuous support during my Ph.D. pursuit. His guidance and invaluable advice helped me in all the time of research and writing of this thesis.

I would also like to thank my Ph.D. qualifying panel members, Dr. CHAN Wing Kwong and Dr. LI Zhenjiang, not only for their precious time and insightful comments on my research reports, but also for the hard questions which inspired me to widen my research from various perspectives.

I am very grateful to my dissertation examiners, Dr. KWOK Lam For, Dr. LI Minming, and Dr. LAI Richard Y, for their invaluable time spent on carefully reading my thesis, and for providing me with precious suggestions and constructive comments to help improve the quality of this thesis.

My sincere thanks also goes to my friends and lab-mates at City University of Hong Kong, Ms. Yan Xiao, Mr. Solomon Mensah, Mr. Kwabena Ebo Bennin, Mr. Shahid Hussain, Mr. Arif Ali Khan, Mr. Jianglin Huang, Mr. Passakorn Phannachitta, and Mr. Xiupei Mei, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years.

Last but not the least, I would like to thank my family for supporting me spiritually throughout writing this thesis. Without their precious understanding and encouragement, it would not be possible for me to complete my dissertation.

Contents

Abstract	i
Acknowledgements	v
1 Introduction	1
1.1 Background	2
1.2 Motivation	4
1.2.1 Problem 1. Data Collection (Code Readability Survey)	4
1.2.2 Problem 2. Model Construction (Manual Feature Engineering)	5
1.3 Contributions of the Thesis	7
1.4 Organization of the Thesis	9
2 Literature Review	11
2.1 Code Readability Research	11
2.1.1 Factors Influencing Code Readability	12
2.1.2 Code Readability Models	14
2.2 Gamification in Software Engineering	17
2.3 Deep Learning in Software Engineering	19

3	GamiCRS: A Gamification System for Code Readability Survey	22
3.1	Preliminaries	23
3.1.1	Intrinsic and Extrinsic Motivation	23
3.1.2	Self-Determination Theory	24
3.1.3	User-Centered Design and Meaningful Gamification	25
3.1.4	Gamification Design and Process	25
3.2	Proposed Approach	27
3.2.1	Identify the Main Objective	27
3.2.2	Identify the Transversal Objective	27
3.2.3	Incentive Mechanism	28
3.2.3.1	Applying GDEs as Extrinsic Motivation	28
3.2.3.2	Applying Crowdsourcing as Intrinsic Motivation	30
3.3	An Implementation of GamiCRS	31
3.3.1	General Overview	31
3.3.1.1	Rating Code Readability	32
3.3.1.2	Uploading Code Snippet	32
3.3.2	Development Framework	34
3.3.3	Code Snippet Repository	35
3.3.3.1	Preparation of the Initial Code Snippet Repository	35
3.3.3.2	Selection Strategy for Next Code Snippet	37
3.3.4	Rules and Regulations of PBL	38
3.3.4.1	Point System	39
3.3.4.2	Badge System	39
3.3.4.3	Leaderboard System	40
3.4	A Field Experiment on GamiCRS	41

3.4.1	Experimental Design	41
3.4.1.1	Participants	41
3.4.1.2	Procedures	42
3.4.2	Experimental Results	43
3.4.3	Post-Experiment Questionnaire	46
3.4.4	Implications	48
3.5	Discussion	51
3.5.1	Gamification Versus Pointsification	51
3.5.2	Existing Problems and Challenges in GamiCRS	52
3.5.3	Threats to Validity	53
3.5.3.1	Internal Validity	53
3.5.3.2	External Validity	54
3.5.3.3	Construct Validity	54
4	DeepCRM: A Deep Learning-Based Code Readability Model	55
4.1	Preliminaries	56
4.1.1	Convolutional Neural Networks	56
4.1.2	Statistical Methods	58
4.2	Proposed Approach	59
4.2.1	Source Code Representation	61
4.2.1.1	Character-Level Representation	62
4.2.1.2	Token-Level Representation	63
4.2.1.3	Node-Level Representation	63
4.2.2	Neural Network Architecture	64
4.2.3	Details of Configuration and Training	65
4.3	Experimental Setup	66

4.3.1	Research Questions	67
4.3.2	Dataset Construction	69
4.3.3	Evaluation Metrics	73
4.4	Results	74
4.4.1	Research Question 1	74
4.4.2	Research Question 2	77
4.4.3	Research Question 3	78
4.4.4	Research Question 4	79
4.5	Exploratory Experiments	80
4.5.1	Exploratory Experiment 1	80
4.5.2	Exploratory Experiment 2	81
4.5.3	Exploratory Experiment 3	82
4.6	Discussion	84
4.6.1	Advantages and Disadvantages of Deep Learning	84
4.6.2	Threats to Validity	87
4.6.2.1	Internal Validity	87
4.6.2.2	Construct Validity	89
5	Conclusions and Future Work	90
5.1	Summary	90
5.2	Future Work	92
A	Acronyms And Abbreviations	95
B	Using Eye Tracking Technology to Analyze the Impact of Stylistic Inconsistency on Code Readability	97
B.1	Context and Problem Statement	98

B.2	Proposed Approach	99
B.2.1	Independent Variables	101
B.2.2	Dependent Variables	101
B.2.3	Secondary Variables	101
B.2.4	Hypothesis	102
B.2.5	Stimuli and Task Design	102
B.2.6	Experimental Procedure	104
C	Stylistic Fingerprinting	106
C.1	Context and Problem Statement	106
C.2	A Taxonomy for Stylistic Metrics	107
C.3	Stylistic Metrics Considered in This Study	109
D	Post-Experiment Questionnaire	113
	Bibliography	115

List of Tables

2.1	Readability Metrics for Natural Languages	12
2.2	Features Used by Buse et al.'s Model	16
3.1	Identify the Transversal Objective	28
3.2	Four Basic Roles of Users in a Crowdsourcing System	30
3.3	Statistical Summary of the Initial Code Snippet Repository	36
3.4	Point System in GamiCRS	39
3.5	Badge System in GamiCRS	40
3.6	Comparison of Mean-Opinion-Scores between the Readable and Unread- able Group	46
3.7	Internal Consistency of the Post-Experiment Questionnaire	48
4.1	Statistical Summary of the Dataset	70
4.2	Effect of the Single Filter Size	75
4.3	Effect of the Multiple Filter Size	76
4.4	Evaluation of the Ensemble Architecture	77
4.5	Average Time and Space Cost per Epoch	79
4.6	Accuracy Achieved by Code Readability Models	85
B.1	Dependent Variables and the Corresponding Measurements	103

C.1 Stylistic Metrics Considered in This Study	110
--	-----

List of Figures

1.1	Readable Code Versus Unreadable Code	2
1.2	Workflow of Code Readability Research	4
2.1	Factors Influencing Code Readability	13
3.1	Gamification Design and Process	26
3.2	A Novel Dual Role Setting in GamiCRS	31
3.3	Readability Rating Page in GamiCRS	33
3.4	A Crowdsourcing Feedback in GamiCRS	34
3.5	Samples from the Initial Code Snippet Repository	37
3.6	Comparison of Inter-Rater Reliability between the Treatment and Control Group	45
3.7	Responses to the Post-Experiment Questionnaire	49
4.1	Architecture of a Typical ConvNet	57
4.2	General Overview of the Proposed Approach	60
4.3	Different Strategies for Source Code Representation	61
4.4	Architecture of Separate ConvNets	65
4.5	Ensemble Neural Network Architecture	66
4.6	The Distribution of Data Sources	73

4.7	Effect of the Number of Feature Maps	76
4.8	Comparison between DeepCRM and the Traditional Code Readability Models	78
4.9	An Auxiliary Input for Model Training	81
4.10	A Modified Inception Module	83
5.1	A Novel Framework for Code Readability Classification	93
B.1	Style Variations	100

Chapter 1

Introduction

Reading source code is a frequent activity in software maintenance and evolution. Developers must read and fully understand source code before attempting any changes. It is therefore desirable to have *readable code* that is less erroneous [42], more reusable [99], and easier to maintain [108]. As shown in Figure 1.1, the visual appearance of a source code in general is referred to as its readability [28].

Research on code readability is of great importance. If automated readability metrics were available, they could be integrated into real-world scenarios such as modern IDEs (Integrated Development Environments) and version control products, and thus assist developers to identify poorly written code [17, 89]. However, as of today, only a few code readability models have been proposed in the literature [17, 42, 99].

This thesis is devoted to the study of code readability. In this chapter, we first briefly introduce the background of code readability research. Then we provide the motivation and contribution of our work.

```

/**
 * Add one zero if necessary
 * @param number
 * @return
 */
private CharSequence addZero(int number) {
    StringBuilder builder = new StringBuilder();

    if (number < 10) {
        builder.append('0');
    }

    builder.append(Integer.toString(number));
}

```

Readable Code (Mean Readability Score - 4.02)

```

/**
 * Creates a new <code>DisbandUnitAction</code>.
 *
 * @param freeColClient The main controller object for the client.
 */
DisbandUnitAction(FreeColClient freeColClient) {
    super(freeColClient, "unit.state.8", null, KeyStroke...
    putValue(BUTTON_IMAGE, freeColClient.getImageLibrary()...
    ImageLibrary.UNIT_BUTTON_DISBAND,
        0));
    putValue(BUTTON_ROLLOVER_IMAGE, freeColClient...
    getUnitButtonImageIcon(
        ImageLibrary.UNIT_BUTTON_DISBAND, 1));
}

```

Unreadable Code (Mean Readability Score - 2.01)

Figure 1.1: Readable Code Versus Unreadable Code

1.1 Background

Software development is a complex activity that requires a group of developers working together. When collaborating, it is necessary for each developer to maintain a high level of code readability, in an attempt to minimize the time and effort others have to spend reading and understanding it. Actually, about 70% of the lifecycle cost of a software project is in the maintenance phase [13]. Estimates show that this high maintenance cost correlates strongly with the difficulty of reading and understanding source code, particularly that

written by others [44, 15, 115], which highlights the importance of a good measurement of code readability.

Code readability refers to a human judgment of how easily a program's source code can be read and understood [17], which has significant relation with software reusability, reliability, maintainability, complexity, and portability metrics [108]. The process of classifying a piece of source code as either readable or unreadable is referred to as *Code Readability Classification*.

In a survey of the information needs of 110 developers and managers at Microsoft, 90% of participants recorded that they would use readability metrics if they were available [18], but little relevant research has been reported. For instance, Buse et al. constructed a code readability model based on a simple set of local code features and showed that it could be 75% to 80% effective at predicting human readability judgments [17]. Making use of the well-known size and Halstead metrics, Posnett et al. [89] presented another model that was simpler and performed better than that of Buse et al.

According to the definition, code readability is essentially an intuitive concept that is claimed to connect with various factors such as the structural pattern (e.g., nested loops and recursive functions) and the spatial layout (e.g., blank lines and indentations) [108, 98]. Because a source code that is considered readable by one person may not be considered so by another, a large-scale survey involving multiple human annotators is necessary to determine whether a source code is readable or unreadable. Therefore, the first phase of modern code readability studies is usually a survey process, followed by a second phase that aims to construct a classification model (a machine learning classifier) based on the dataset gathered during the first phase as the ground truth. The entire workflow is illustrated in Figure 1.2.

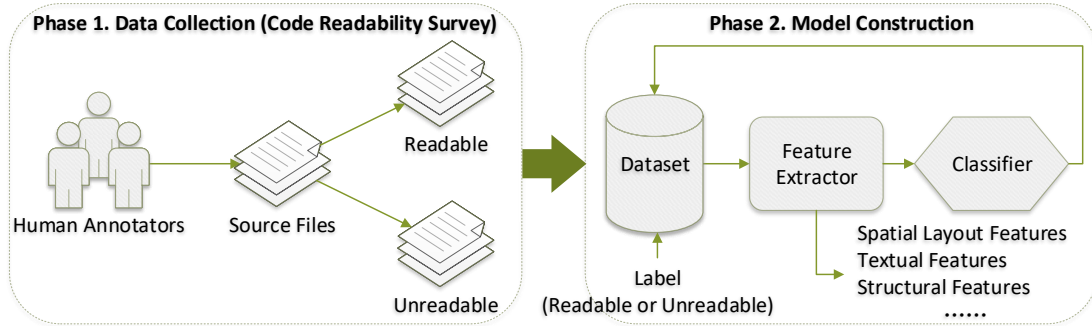


Figure 1.2: Workflow of Code Readability Research

1.2 Motivation

Despite the encouraging results, several problems limit the performance and generality of the existing approaches. One major obstacle is the difficulty of experimentally reifying people’s subjective perceptions of whether a source code is readable [89]. Another significant threat is likely the widely used feature engineering process. We describe the two problems in detail below.

1.2.1 Problem 1. Data Collection (Code Readability Survey)

Code readability is an intuitive concept, a source code that an individual perceives as clear and clean may not be considered equally by others. Accordingly, a large-scale survey involving multiple domain experts is fundamentally necessary to abstract human perceptions into consistent, standardized representations.

Although survey plays a critical role in code readability studies, its methodology has long been criticized for the dullness and pointlessness. For researchers, a survey is generally too complicated to design, too laborious to perform, and too difficult to recruit human

participants [19]. Meanwhile, as the traditional survey process is in itself neither amusing nor rewarding, it is possible that participants remain low incentives and motivation to contribute, resulting in negative behaviors such as premature termination and random responding [43, 51]. These problems inspire our research into finding effective ways to foster voluntary participation and active engagement, with the underlying goal of improving the quality of survey results.

Gamification is the application of typical game design elements (e.g. points, levels, and badges) into non-game contexts [37], which is a promising approach for motivating desired behaviors. When employed properly, it can be a very powerful tool to improve engagement and enhance productivity. Successful implementations range from education [35], business [111], work [62], to health [73]. A literature review has shown that the majority of empirical studies on gamification produce positive effects and benefits [50]. Referring to the success of previous studies, we envision the use of game-based mechanics and aesthetics in the context of code readability survey as a persuasive technique.

1.2.2 Problem 2. Model Construction (Manual Feature Engineering)

As shown in Figure 1.2, a common practice in the second phase of code readability research is to handcraft readability-related features, and then apply them as predictors to differentiate between readable and unreadable codes. For instance, Buse et al. [17] designed a set of layout features (e.g., the number of blank lines¹), whereas Scalabrino et al. [99] attached greater importance to textual aspects (e.g., the readability of comments). However, this widely used feature engineering method has the following problems:

- **Inefficient**

¹In fact, several recent studies suggested that simple blank lines are more important than comments to local judgments of code readability [17, 110].

- Handcrafting good features is difficult and time-consuming because it requires strong domain-specific knowledge [29]. Moreover, researchers and practitioners must examine the validity of all promising features separately or as a combination, which makes the manual approach expensive or even impossible.

- **Ineffective**

- First, the process can capture only partial information about the source code, which may not be adequate. For instance, Buse et al. [17] proposed the first model of code readability based on a set of layout features (e.g., the number of blank lines). We cannot claim that the features proposed by Buse et al. are sufficient to represent code readability, and this is indeed the case. In a recent study, Scalabrino et al. [99] further introduced a set of textual features by considering lexical aspects of source code involved in identifiers and comments. The results showed that the proposed features complemented Buse et al.’s work when predicting code readability judgments. Similarly, we still cannot claim that the features proposed by Buse et al. and Scalabrino et al. are sufficient to represent code readability.
- Second, features that are usable for one dataset may not be usable for another. For instance, on Buse et al.’s Dataset, Buse et al.’s Model [17] can achieve the best performance (in comparison with Dorn et al.’s Model [42], Posnett et al.’s Model [89], and Scalabrino et al.’s Model [99]), whereas on Dorn et al.’s Dataset, Buse et al.’s Model performs unsurprisingly worse than Dorn et al.’s Model [42].
- Third, redundancies and overlaps may exist among the manually designed features, which is likely to limit the model’s performance. For instance, Buse et

al. [17] designed a total of 25 features to represent code readability. However, Posnett et al. [89] showed that a much simpler model with only three features can outperform Buse et al.'s Model. On the other hand, the feature selection method has been used by several previous studies [41, 99] to identify a subset of key features, which also indicates that not all manually designed features are relevant and have an impact on code readability.

Although programs usually contain statistical properties, they are difficult for humans to capture [53, 88]. To address this issue, we turn to Convolutional Neural Networks (ConvNets). The reasons behind using ConvNets are as follows: 1) ConvNets include the feature extractor in the training process rather than requiring it to be manually designed [57]. 2) ConvNets achieve remarkable results for tasks that deal with inputs in the form of multiple arrays (e.g., image recognition). Inspired by this, we treat a source code as a matrix (a two-dimensional array) of symbols. We expect that ConvNets can automatically form a deep understanding of what constitutes a readable code.

1.3 Contributions of the Thesis

The main contributions of this thesis are as follows:

- Based on the Self-Determination Theory and User-Centered Design, we propose a gamification system for code readability survey (GamiCRS), in which a novel incentive mechanism is presented to help foster real engagement.
- We implement GamiCRS as a web-based application to explore its possibilities and limits. As compared to prior work, we contribute a more detailed implementation process in a systematic manner.

- We carefully plan a between-subjects design to quantitatively reflect to what extent GamiCRS impacts participants' performance in terms of productivity and quality.
- Based on the Technology Acceptance Model (TAM), a questionnaire approach is utilized to investigate participants' impressions and attitudes toward GamiCRS. We are the first to employ TAM in the field of gamified surveys.
- The proposed gamification system has been proven to be effective for enhancing code readability survey regarding user experience and data quality, which is supported by a sound theoretical and experimental evidence. We believe that our study can serve as a significant reference for future development of gamification applications in the context of software engineering research.
- A simple and customizable representation strategy (with three different granularities) is introduced to convert source codes into integer matrices as the input to ConvNets. The method enables deep learning-based program analyses, which can be easily applied to other software engineering activities.
- DeepCRM, a deep learning-based framework that integrates three separate ConvNets (corresponding to the three granularities), is proposed for code readability classification. Our approach eliminates the need for manual feature engineering and can learn various levels of features automatically from the source code.
- A new dataset, which contains more than 25,000 code snippets retrieved from open source Java projects, is released to support future code readability studies.²
- We compare the performance of DeepCRM with five state-of-the-art code readability models, namely, those of *Buse et al.* [17], *Posnett et al.* [89], *Dorn et al.* [42],

²<https://github.com/CityU-QingMi/DeepCRM>

Scalabrino et al. [99], and *A Comprehensive Model* [99]. The experimental results show that DeepCRM outperforms the best competitor, reaching 83.8% accuracy and 83.5% f-measure. Although deep learning has achieved remarkable success in other areas, to the best of our knowledge, we are the first to observe its effectiveness in code readability classification.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows.

In Chapter 2, we first review related work on code readability research, including factors influencing code readability and previously reported code readability models. Then we briefly describe deep learning and gamification applications in the context of software engineering.

Based on the Self-Determination Theory and User-Centered Design, we introduce a new interactive method in Chapter 3 and implement it as GamiCRS, an online gamification system for code readability survey. We assume that the incorporation of gamification techniques and infrastructure can produce psychological (e.g., satisfaction and fun) and behavioral (e.g., participation and performance) benefits for survey participants. To scientifically test this hypothesis, a field experiment is conducted in a real-life environment to compare GamiCRS with its non-gamified counterpart. The empirical results validate the positive effects of GamiCRS on the performance of survey participants (with more numerous and higher quality readability ratings).

Chapter 4 describes DeepCRM in detail. Unlike previous studies (based on traditional machine learning), we introduce a completely new method for code readability classification using deep learning techniques. Our approach has the advantage of requiring no feature

engineering (one of the most time-consuming parts of traditional machine learning). We evaluate our approach against five state-of-the-art code readability models. The experimental results show that DeepCRM can outperform traditional models. The improvement in accuracy ranges from 2.4% to 17.2%, which suggests the promising future of applying deep learning techniques to the field of code readability classification.

We conclude the thesis in Chapter 5, which includes the results that have been drawn from the research, main contributions, and suggestions for future work.

Chapter 2

Literature Review

Code readability is one of the most important software quality attributes. When a developer initially writes the code, his/her knowledge of the code is very detailed. However, a year later no one is going to know the code that well again. The developer has to understand how the code works by reading it. If the code is poorly written, it will be difficult to understand, longer to debug, hard to maintain, and tricky to extend. It is therefore desirable to have readable code. However, there has been a little previous work on this special area.

In this chapter, we first describe related work on code readability research. Then we discuss deep learning and gamification applications in the context of software engineering.

2.1 Code Readability Research

As shown in Table 2.1, a wide variety of readability metrics for natural languages are available in the literature (e.g., the Flesch-Kincaid Grade Level [45] and the SMOG Grading [72]), yet only a few studies have considered the counterpart targeted in the context of program source code.

Table 2.1: Readability Metrics for Natural Languages

Metric	Description ^a
ARI	$Automated\ Readability\ Index = 0.5 \times \frac{W}{S_r} + 4.71 \times \frac{C}{W} - 21.43$
SMOG	$Simple\ Measure\ of\ Gobbledygook = 1.043 \times \sqrt{W_{3Sy} \times \frac{30}{S_r}} + 3.1291$
Flesch	$Flesch\ Reading\ Ease = 206.835 - 1.015 \times \frac{W}{S_r} - 84.6 \times \frac{S_y}{W}$
GunningFog	$Gunning\ Frequency\ of\ Gobbledygook = 0.4 \times \left(\frac{W}{S_r} + \frac{100 \times W_{3Sy}}{W} \right)$
FleschKincaid	$Flesch\ Kincaid\ Grade\ Level = 0.39 \times \frac{W}{S_r} + 11.8 \times \frac{S_y}{W} - 15.59$
FORCAST	$FORCAST = 20 - \frac{W_{1Sy} \times \frac{150}{W}}{10}$
ColemanLiau	$Coleman\ Liau = 5.88 \times \frac{C}{W} - 29.6 \times \frac{W}{S_r} - 15.8$

^a S_r stands for the number of sentences, W for the number of words, C for the number of characters, S_y for the number of syllables, W_{1Sy} for the number of words with exactly one syllable, W_{3Sy} for the number of words with at least three syllables.

Specifically, prior work related to code readability can be broadly divided into two categories:

- Investigation of factors that can affect code readability.
- Derivation of general models to represent code readability.

We describe the two directions in detail below.

2.1.1 Factors Influencing Code Readability

Many earlier studies focused on exploration of the influential factors of code readability, such as those shown in Figure 2.1. In this section, we briefly discuss these factors.

Binkley et al. [10, 11, 12] conducted an empirical study with 135 subjects (consisting of programmers and non-programmers) to analyze the effects of identifier naming conventions (i.e., camelCase and under_score) on code readability. The underlying hypothesis is that identifier style affects the speed and accuracy of program comprehension. To investigate the hypothesis, Binkley et al. recorded the time and visual effort subjects needed to detect a particular identifier (from four choices). The results indicated that camelCase was easier

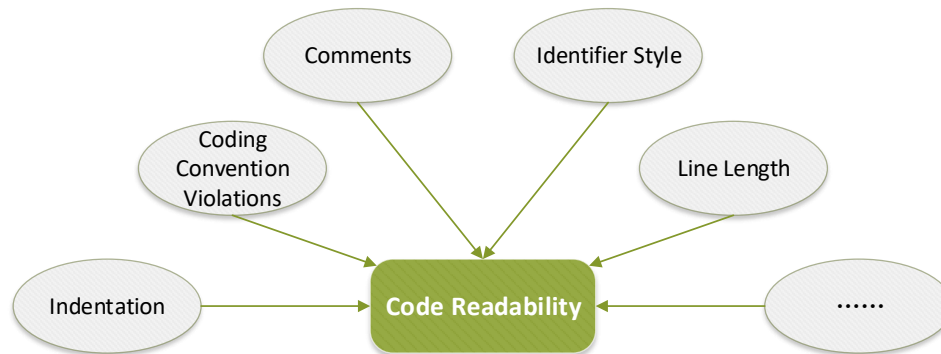


Figure 2.1: Factors Influencing Code Readability

to read than under_score variables,¹ especially for beginning programmers, whereas expert programmers appeared to not be impacted greatly by identifier style.

Sasaki et al. [97] proposed a reordering technique to improve code readability by shortening the distance between the definition of a variable and its reference. Specifically, two reordering strategies are applied: 1) moving statements into inner blocks to narrow down variable scope; 2) moving statements close to each other within the same block. An experiment with 44 subjects confirmed that the reordered methods had better readability than their originals.

Using five open source projects, Lee et al. [66] tested whether significant violations of coding conventions affected the readability of developed codes. Specifically, they counted the total number of rule violations in each file and tested the difference between the two sample groups of more/less readable codes using Wilcoxon's Rank-Sum test. The results showed that the less readable group had more rule violations and the difference was statistically significant at a 95% confidence level.

¹However, an eye-tracking replication of Binkley et al.'s study indicated otherwise [101].

Tashtoush et al. [108] developed a novel approach to examine the level of impact the 25 proposed code features have on readability. A survey was distributed to a random number of expert programmers to weight each feature based on their knowledge or experience. The results showed that most of proposed features have positive impact on code readability including: meaningful names, consistency, and comments, whereas fewer features such as arithmetic formulas, nested loops, and recursive functions showed to have a negative impact.

Wang et al. [110] proposed an automatic blank line insertion algorithm to separate meaningful blocks, primarily because vertical spacing is believed to help code readability. In particular, they leverage both program structure and naming information to identify logically related code segments. The results from two human studies indicated that the automatically generated blank lines are accurate, which can match the majority of human opinions of where vertical spacing should be used.

As discussed above, considerable research efforts have been devoted to identifying the influential factors of code readability. However, there is little sound and solid evidence of how and to what extent can these factors impact the readability of the source code. To bridge the research gap, we propose an empirical experiment in which eye tracking technology is introduced to quantitatively reflect subjects' cognitive efforts and mental processes [74]. We use stylistic inconsistency (one of the factors influencing code readability) as an example. The details are shown in Appendix B.

2.1.2 Code Readability Models

Alternatively, a few studies attempted to model code readability using combinations of surface-level features.

Aggarwal et al. [1] proposed a new factor CR (Comments Ratio) to estimate the readability of source code in a study of software maintainability: $CR = LOC/LOM$, where LOC refers to total lines of code and LOM refers to total lines of comments. The lower the ratio, the better is readability.

In our previous work regarding stylistic fingerprinting [75], we proposed a set of metrics to represent code readability. We denote these metrics as PRM (Programming Readability Metrics). Making use of PRM, together with PFM (Programming Format Metrics) and PLM (Programming Language Metrics), we implemented a fingerprinting methodology to characterize the programming style of an individual or a single document.²

Analogous to the Flesch Reading Ease Score (a widely used readability metric for English text) [45], Börstler et al. [14] proposed SRES (Software Readability Ease Score) to judge code readability according to the average length of lexemes (identifiers, keywords, and symbols) and the average number of words per statement/block (delimited by curly brackets and semicolons). The experimental results showed that SRES is less sensitive to commenting and whitespace compared with other readability measures.

Buse et al. [20, 17] used data collected from 120 human annotators to construct the first model of code readability based on a set of local code features, as shown in Table 2.2. Moreover, they found that the readability metric correlated strongly with three measures of software quality: code changes, automated defect reports, and defect log messages.³ Buse et al.'s work was a major contribution to this area because it opened up the possibility of automated readability metrics and provided an excellent platform for conducting future readability experiments.

²The stylistic metrics adopted in our previous work [75] are provided in Appendix C.

³To test the hypothesis that readability correlates with external conventional metrics of software quality, Buse et al. used McCabe's Cyclomatic measure (an indicative example of an automated model of code complexity) and FindBugs (a popular static bug-finding tool).

Table 2.2: Features Used by Buse et al.'s Model

Feature	Avg.	Max.
line length (# characters)	+	+
# identifiers	+	+
identifier length	+	+
indentation (preceding whitespace)	+	+
# keywords	+	+
# numbers	+	+
# comments	+	
# periods	+	
# commas	+	
# spaces	+	
# parenthesis	+	
# arithmetic operators	+	
# comparison operators	+	
# assignments (=)	+	
# branches (if)	+	
# loops (for, while)	+	
# blank lines	+	
# occurrences of any single character		+
# occurrences of any single identifier		+

Using the same dataset from [17], Posnett et al. [89] presented a simple, intuitive theory of code readability that relied on two main measures: size and code entropy (see equation below). They showed that the simple model outperformed Buse et al.'s model across all performance measures (e.g., F-Measure and ROC).

$$z = 8.87 - 0.033V + 0.40Lines - 1.5Entropy \quad (2.1)$$

In order to build a model of code readability that is more likely to generalize than previous work, Dorn et al. [42] performed a large-scale human study involving over 5000 participants and proposed additional structural pattern features, visual perception features,

alignment features, and natural language features. They showed that the automatic metric agrees with human judgments as well as they agree with each other and better than previous work.

In a recent study, Scalabrino et al. [99] further introduced a set of textual features that were based on source code lexicon analysis.⁴ The results indicated that the proposed features complemented classic structural features when predicting code readability judgments.

As discussed above, existing studies still focus on handcrafting features from different aspects that intuitively seem to correlate with code readability, and then exploring various machine learning algorithms based on the newly proposed features. The manual feature engineering process is neither effective nor efficient. In order to eliminate the need for manual feature engineering and increase the accuracy of code readability classification, we introduce a deep learning-based approach that can learn complicated features automatically from the source code.

2.2 Gamification in Software Engineering

Over the past decade, the relation between gamification and software engineering (SE) is on the rise.

Given that many activities in SE are typically challenging and seldom amusing, gamification has been widely applied as a compelling incentive for supporting all phases of SE lifecycle.

Snijders et al. [104] introduced REfine, a gamified online platform for eliciting and refining requirements by involving a crowd of stakeholders: users, developers, analysts,

⁴Seven textual features of source code are proposed in [99] to help characterize readability: Comments and Identifiers Consistency, Identifier Terms in Dictionary, Narrow Meaning Identifier, Textual Coherence, Comments Readability, and Number of Meanings.

etc. A case study showed the potential of REfine for improving requirements engineering in software production.

Singer et al. [102] gamified version control system using mechanisms of social software to encourage more frequent commits. After 37 students used the application in a semester-long cooperative development project, they found a balance of positive and negative comments.

Minelli et al. [78] presented the initial design and implementation of a micro-gamification layer on top of an object-oriented IDE to reflect the deeds of the developers, which aimed to praise them in case the development session was productive.

Meanwhile, a large number of gamification research have appeared in the domain of SE education.

Xie et al. [116] illustrated Pex4Fun, an interactive gaming-based teaching and learning platform for .NET programming languages. Their illustration of Pex4Fun focused on the gaming, social dynamics, educational usage, and software engineering technologies, being four common aspects of a typical project on educational software engineering.

Roungas et al. [93] presented a conceptual model of educational games, which defined the structure of the game design document and provided a common ground that could minimize misunderstandings among heterogeneous stakeholders. The preliminary evaluation, which involved interviews with serious games experts, has shown a positive attitude towards the potential benefit of a web, model-driven environment to mitigate the problems of GDD-based game design.

Arawjo et al. [4] developed Reduct, an educational game embodying a comprehension-first approach to teaching programming with gamified semantics. A lab study demonstrated that novices can learn core concepts expressed in actual JavaScript in a short timeframe by playing Reduct.

However, there is limited literature available regarding gamification for SE surveys. Worth to mention is the work of Biegel et al. [9], which implemented a gamification approach as a Facebook plugin to make code tagging more fun. A pilot user study suggested that the game mechanics were motivating and promoted the proposition of reasonable tags. To the best of our knowledge, gamification has not been included in the field of code readability survey. Referring to the success of previous studies, we exemplify how gamification mechanisms can be leveraged to improve the survey procedure subject to the context of code readability research.

2.3 Deep Learning in Software Engineering

Deep learning has been proven to be a very powerful method in a variety of fields such as image recognition [60, 107] and natural language processing [25, 77]. It has also attracted considerable attention from researchers and industrial practitioners in the software engineering community. Successful applications include effort estimation [23], defect prediction [109], bug localization [63], code clone detection [113, 112], and API recommendation [49].

Dam et al. [29] presented a vision for DeepSoft, a generic framework built upon Long Short-Term Memory for modeling software and its development process to predict future risks and recommend interventions. The deep learned patterns of software could be used to address a range of challenging problems such as code and task recommendation and prediction.

Wang et al. [109] leveraged Deep Belief Networks to automatically learn semantic features of programs to improve defect prediction. The evaluation on ten open source projects showed that the automatically learned semantic features can significantly improve both

within-project (WPDP) and cross-project defect prediction (CPDP) compared to traditional features. For WPDP, the semantic features achieved an average improvement of precision by 14.7%, recall by 11.5%, and F1 by 14.2%. For CPDP, the semantic feature based approach outperformed the state-of-the-art technique TCA+ built on traditional features by 8.9% in F1.

Choetkietikul et al. [23] developed a prediction system for estimation of story points based on a novel combination of Long Short-Term Memory and Recurrent Highway Network. An empirical evaluation showed that the proposed approach consistently outperformed three common baselines and two alternatives in both Mean Absolute Error and the Standardized Accuracy.

Gu et al. [49] proposed DeepAPI, a deep learning-based method to generate relevant API usage sequences for a given natural language query. The empirical study showed that the proposed approach generated largely accurate API sequences and outperformed the related approaches.

White et al. [113, 112] introduced learning-based detection techniques to represent code fragments for code clone detection using Recurrent Neural Networks and Recursive Neural Networks.

Lam et al. [63] constructed HyLoc, a model that combines the revised Vector Space Model (an advanced IR technique) with Deep Neural Networks to recommend potentially buggy files for a bug report. The empirical evaluation on several real-world projects showed that DNN and IR complement well to each other to achieve higher bug localization accuracy than individual models.

To the best of our knowledge, deep learning has never been used to improve code readability studies. Inspired by the aforementioned work, we explore the possibility and feasibility of applying Convolutional Neural Networks to the field of code readability classification.

Chapter 3

GamiCRS: A Gamification System for Code Readability Survey

Code readability survey plays a critical role in abstracting people’s perceptions of whether a source code is readable. However, its methodology has long been criticized for the dullness and pointlessness, inevitably leading to negative behaviors that can be attributed to the lack of motivation of survey participants. To address this problem, we incorporate game-based mechanics into the context of code readability research, which is to make participants feel positive when engaging in the survey process, with the underlying goal of improving the quality of survey results. Based on the Self-Determination Theory and User-Centered Design, we propose a novel gamification system (GamiCRS), in which a complete incentive mechanism is constructed with the combination of both intrinsic and extrinsic motivators.

To ensure its efficacy, we carefully plan a field experiment to compare GamiCRS with its non-gamified counterpart. The empirical results validate the positive effects of GamiCRS on the performance of survey participants (with more numerous and higher quality readability ratings). As many activities in software engineering are typically challenging

and seldom amusing, gamification can be applied as a compelling incentive supporting a wide variety of possible applications in order to bring the same attitudinal and behavioral changes.

The rest of this chapter is organized as follows. Section 3.1 presents the related work and the theoretical foundations. Section 3.2 proposes a gamification system for code readability survey and Section 3.3 implements it. In Section 3.4, we perform a field experiment to explore how effective the gamification system is. A discussion is provided in Section 3.5.

3.1 Preliminaries

We begin with a brief introduction of several theories specifically in human motivation drawn from the domain of psychology, which serve as the theoretical foundations supporting the overall design of our gamification system.

3.1.1 Intrinsic and Extrinsic Motivation

Motivation is the process that initiates and energizes goal-oriented behaviors, which is the core element driving individuals to accomplish a certain task [56]. Motivation varies not only in amount (i.e., how much motivation), but also in nature (i.e., what type of motivation) [94], which can be divided into two categories:

- **Intrinsic Motivation:** the doing of an activity for its inherent satisfactions (e.g., altruism, competition, cooperation, sense of belonging, love, and aggression [80]), which occurs when an individual enjoys the activity or regards it as an opportunity to explore, learn, or actualize the potentials [26].

- **Extrinsic Motivation:** the doing of an activity for some separable consequences (e.g., financial incentives, classifications, levels, points, badges, awards, and missions [80]), which can be used to inspire an individual's interest to pursue a behavior.

Research has shown that people's performance varies according to whether his/her motivation stems from intrinsic or extrinsic. Generally, intrinsic motivation offers intense, long-term engagement and results in high-quality behaviors, whereas extrinsic motivation is relatively less effective yet indispensable in the scenario that the activity is in itself neither engaging nor rewarding to the individual, some external rewards are thus introduced as a great way to foster participation, with an underlying goal of developing people's real interest in the course of being involved in a certain activity [26].

3.1.2 Self-Determination Theory

Self-Determination Theory (SDT), proposed by Ryan and Deci [95], is the most well-known theory to identify factors that facilitate or undermine people's self-motivation and personality integration. The hypothesis is that individuals have three basic psychological needs that when fulfilled facilitate intrinsically motivated behaviors:

- **Competence:** a need for challenge or feelings of effectance (e.g., opportunities to acquire new knowledge and skills, to receive positive feedback, or to be optimally challenged).
- **Relatedness:** a need to enrich relationships with others.
- **Autonomy:** a need to feel a sense of volition or willingness when doing a task.

These needs are universal, irrespective of age, gender, and culture. According to SDT, human beings can be proactive, creative, and engaged when activities are performed out of their intrinsic motivations, such as curiosity, interest, personal value, and so forth [94].

3.1.3 User-Centered Design and Meaningful Gamification

Gamification is a cost-effective approach for promoting participation and behavioral changes, as users endeavor to reach predetermined goals for external rewards [118]. Nonetheless, almost all forms of extrinsic rewards reduce the intrinsic motivation that users have for the activity [33]. Noticing the long-term negative impact, Nicholson et al. [84] proposed the concept of meaningful gamification by integrating disparate theories (e.g., user-centered design [85], organismic integration theory [34], and situated motivational affordance [36]), indicating that users' needs and desires should be put at the center of any gamification project. In other words, if the gamified system can be made meaningful to individuals (e.g., relevant to their backgrounds, interests, or goals), the intrinsic motivation will be improved as there is less dependence upon external rewards.

According to Nicholson et al. [84], meaningful gamification should focus on users' long-term benefits and should result in positive change in users' behavior and mindset. It is suggested that, during the gamification process, every decision should be made for a central concern: "How does this benefit the user?".

3.1.4 Gamification Design and Process

Gamification is defined as the application of typical game design elements (GDEs) into non-game contexts [37]. Accordingly, all gamified systems share three essential elements: users, a non-game task, and a set of GDEs that motivate users to perform the task. Interconnecting these elements, Wilson et al. [114] proposed a relationship-based model to guide the gamification design, which is shown in Figure 3.1a. They claimed that a successful gamification relies on the relationships between the elements:

- **Motivation (User-GDE Relationship):** motivation is afforded when GDEs satisfy users' basic psychological needs of competence, relatedness, and autonomy.
- **Meaning (GDE-Task Relationship):** as tasks are not created equal, the most appropriate GDEs are those that align with the nature of the task.
- **Valuation (User-Task Relationship):** the more value the users put on the task, the more effective the gamification will be.

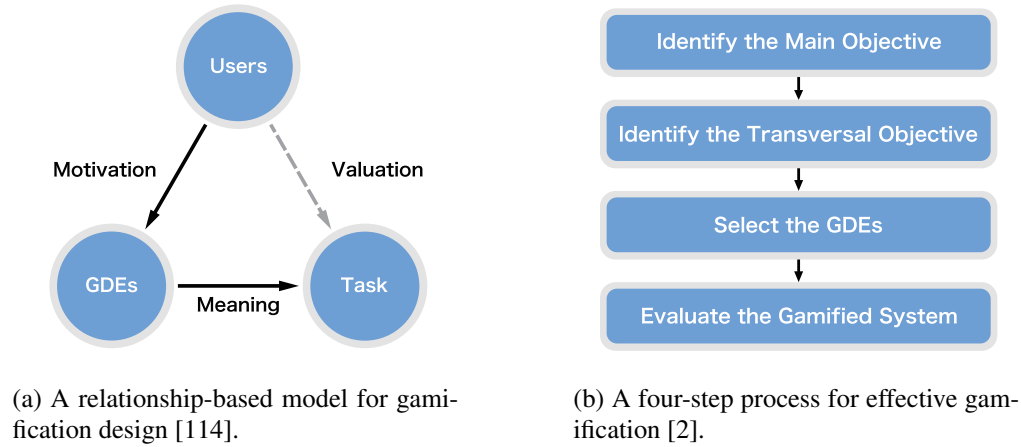


Figure 3.1: Gamification design and process.

Consistent with Wilson et al.'s model, Aparicio et al. [2] proposed a four-step process for developing the effective gamification on the basis of Self-Determination Theory, which is demonstrated in Figure 3.1b. The rest of this research, i.e., the design (Section 3.2), the implementation (Section 3.3), and the evaluation (Section 3.4) of GamiCRS, follows Aparicio et al.'s four-step gamification process.

3.2 Proposed Approach

In this section, we present the design process of GamiCRS based on the psychological theories as introduced in Section 3.1. Note that what we are trying to do is not just rewarding individuals for their contributions with game points, such a simple scoring mechanism (also known as pointsification, the details are discussed in Section 3.5.1) is destined to failure [78]. Rather, we intend to propose a systemic incentive mechanism with the combination of both intrinsic and extrinsic motivators that is able to foster real engagement.

3.2.1 Identify the Main Objective

Survey is important in abstracting human perceptions in code readability research. However, the process is generally considered as laborious and tedious. To address this issue, the main objective in Aparicio et al.'s framework (i.e., the non-game task in Wilson et al.'s model) that we intend to gamify is brought forth: "Can we make the code readability survey a much more engaging and rewarding process?".

3.2.2 Identify the Transversal Objective

As the second step, we identify the transversal objective and present the reasons behind the use of gamification mechanisms.

Given that the traditional code readability survey (CRS) is not inherently enjoyable, some extrinsic stimuli (e.g., financial rewards) are often introduced to facilitate participation. For instance, Buse et al. [17] rewarded all survey respondents with \$5 USD (and additional \$5 USD for early starters), whereas no monetary incentives were reported in Scalabrino et al.'s study [99], yet only 9 out of 30 invited students completed the assessment.

Unlike previous studies, we intend to activate survey participants with the help of various game-based mechanics and aesthetics. Specifically, Table 3.1 enumerates a range of problems with CRS, and how gamification can fix them (i.e., the transversal objective in Aparicio et al.’s framework).

Table 3.1: Problems with code readability survey (CRS) and how gamification can fix them (i.e., identify the transversal objective).

Problem with CRS	How Gamification Can Fix the Problem
CRS is unrewarding.	Games can help users feel more rewarded through continuous and prompt feedback for achieving given tasks, which is of great importance to keeping users on course.
CRS is pointless.	Games can provide users with clearer missions (i.e., long- and short-term goals), which helps users focus attention.
CRS is isolating.	Games can help users band together through collaboration and friendly competition.

3.2.3 Incentive Mechanism

To foster voluntary participation and active engagement, we propose a complete incentive mechanism based on the Self-Determination Theory and User-Centered Design.

3.2.3.1 Applying GDEs as Extrinsic Motivation

Game design elements (GDEs) act as the toolkit for gamification consisting of bits and pieces we work with to build a game [96], which can be used as a great way to motivate certain behaviors. According to Aparicio et al.’s framework, our next step is to select GDEs that simultaneously match the objectives and support the basic psychological needs of users.

The best gamifications are not the ones that incorporate all of the elements, but those choose the elements most properly [96]. Corresponding to the transversal objective identified in Table 3.1, we adopt the classic PBL triad (Points, Badges, and Leaderboards) [111]

in GamiCRS to reward users through different dimensions. The primary reasons are as follows: 1) PBL is applied in most gamification projects as they are simple (most users have no difficulty in understanding their rules and regulations) yet significantly effective. 2) Although a highly interactive, game-like application often produces a better user experience, it requires a lot more time and effort to implement [52]. PBL is employed as a reasonable compromise.

- **Points:** a way of providing quick feedback in GamiCRS. As rewarding users with points for their actions may help to change, or boost certain attitudes and behaviors, we equate the core activities in code readability survey with points, which can hopefully motivate users to contribute more.
- **Badges:** a visual representation of achievements in GamiCRS. We reward users with badges for reaching specified thresholds, accomplishing particular tasks, and so forth. Badges can offer guidance as to what is possible within the system, and most importantly, to provide long- and short-term goals for users to strive toward [111], which have been confirmed to have positive effects on motivation and engagement [35, 52].
- **Leaderboards:** a scoreboard displaying the ranking of the leading users in GamiCRS. The presence of a leaderboard can bring users' achievements forward, which is a powerful motivator to trigger friendly competition [71].

Note that the introduction of GDEs aims to fulfill users' basic psychological needs according to SDT, but this motivation is directed toward GDEs rather than the task itself. In other words, the intrinsically motivating GDEs are essentially extrinsic motivators with regard to the non-game task [114].

3.2.3.2 Applying Crowdsourcing as Intrinsic Motivation

Extrinsic motivation can absolutely strengthen engagement, yet it should not be overused, for the reason that excessive external rewards might deteriorate individuals spontaneous interest for the activity [26]. In many cases, intrinsic motivation should be encouraged.

Code readability is one of the important software quality attributes that novice programmers need to learn in their programming classes, unfortunately most of the students do not have the necessary work experience or background to appreciate the importance of code readability. While we always strive to activate users' intrinsic motivation, we facilitate this by incorporating a crowdsourcing approach into GamiCRS. That is, users of GamiCRS (especially students and novice programmers) are able to receive judgments from experienced colleagues or trusted peers on whether their source code is readable.

As is shown in Table 3.2, Doan et al. [40] identified four basic roles of human users in a crowdsourcing system. Note that throughout the course of a typical survey, users often play a single role (i.e., *Perspective Providers*), whereas we propose a novel dual role setting in GamiCRS (Figure 3.2). Specifically, not only can users evaluate readability level of prepared code snippets as *Perspective Providers*, they are also granted the capabilities of uploading their own code snippets as *Content Providers* and then receiving a crowdsourcing feedback from a multitude of human experts, which makes GamiCRS novel in this regard.

Table 3.2: Four basic roles of users in a crowdsourcing system [40].

Role	Description
Slaves	Humans help solve the problem in a divide-and-conquer fashion, to minimize the resources of the owners.
Perspective Providers	Humans contribute different perspectives, which when combined often produce a better solution.
Content Providers	Humans contribute self-generated content.
Component Providers	Humans serve as components in the target artifact, such as a social network.

Although the dual role setting limits the possible application scope of GamiCRS, it allows the activation of individuals' intrinsic motivation. For researchers, it is a great way to enrich and evolve the code snippet repository, while for users, it is an opportunity to acquire a crowdsourcing judgment, which serves as the integration point of researchers' objectives and users' needs.

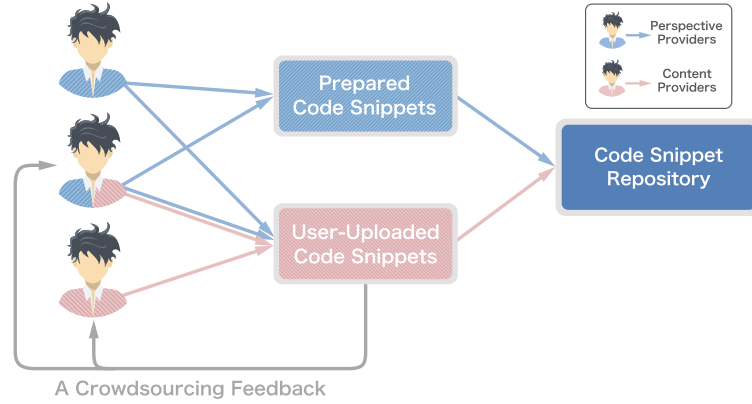


Figure 3.2: A novel dual role setting in GamiCRS.

3.3 An Implementation of GamiCRS

In this section, we implement GamiCRS as a web-based gamification system. We begin by providing a general overview of GamiCRS. After briefly introducing the development framework, we describe the preparation of the code snippet repository subject to evaluation. Finally, we present the rules and regulations specific to the PBL triad (Points, Badges, and Leaderboards).

3.3.1 General Overview

Corresponding to users' dual role setting (Figure 3.2), GamiCRS consists of two main functions: rating code readability and uploading code snippets.

3.3.1.1 Rating Code Readability

Immediately after logging in, users are brought to the main page that provides background information and necessary guidance as to what is possible within the system. Users can then navigate to other parts of GamiCRS, or they can begin to rate the readability degree of a certain code snippet. Figure 3.3 serves as a cohesive picture of the readability rating page, which is composed of four major modules:

- **User Profile** (the top left portion of Figure 3.3): the basic information of a user, for instance, the current points, the acquired badges, and the relative positioning among his/her peers.
- **Points Leaderboard** (the bottom left portion of Figure 3.3): a vertical structure displaying the ranking and the current points of the top 10 leading users.
- **Code Snippet** (the top right portion of Figure 3.3): a piece of source code (rendered with syntax highlighting) subject to evaluation.
- **Subjective Evaluation** (the bottom right portion of Figure 3.3): a five-point Likert scale [67] which measures users' perceptions on how easily the code snippet can be read and understood. In addition, users are allowed to provide feedback in the form of comments (optional).

3.3.1.2 Uploading Code Snippet

To foster intrinsic motivation and sustain engagement in the long run, GamiCRS supports and encourages users to upload their own code snippets. As shown in Figure 3.4, after submitting a piece of source code, users may well receive a crowdsourcing feedback on its readability level.

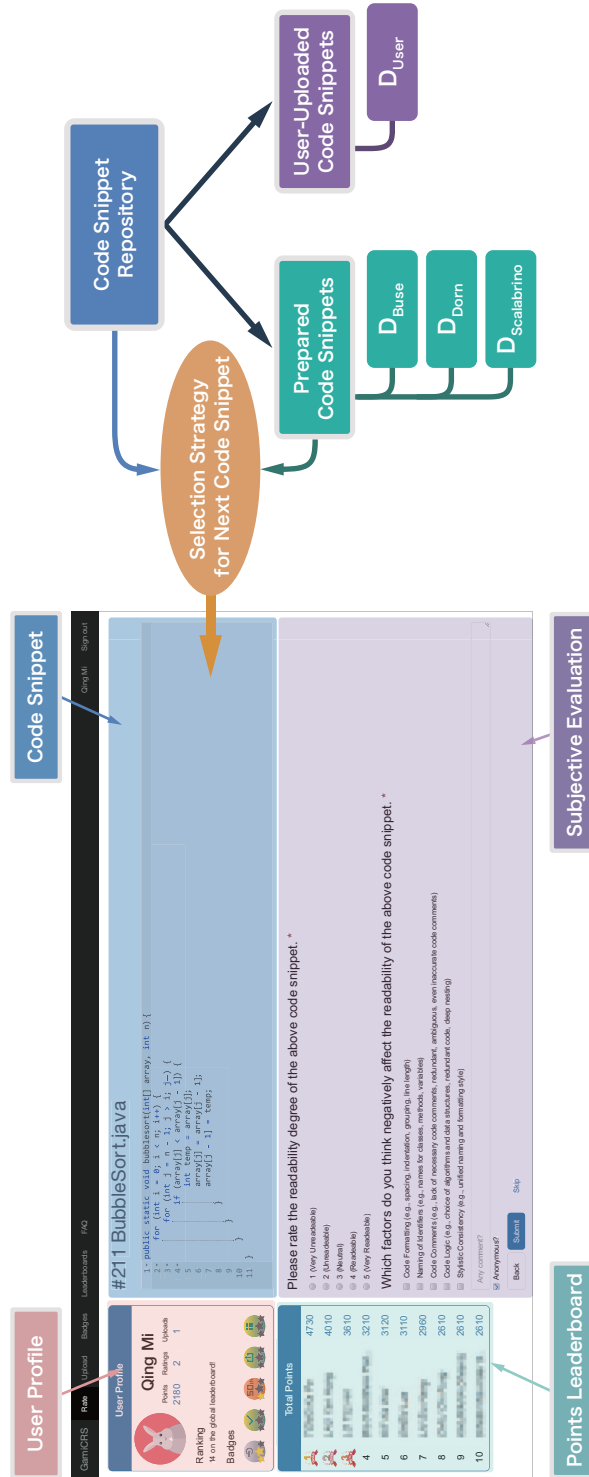


Figure 3.3: Readability rating page in GamiCRS displaying the user profile, the points leaderboard, a certain code snippet and a five-point Likert scale. A selection strategy is proposed to determine the next code snippet subject to evaluation.

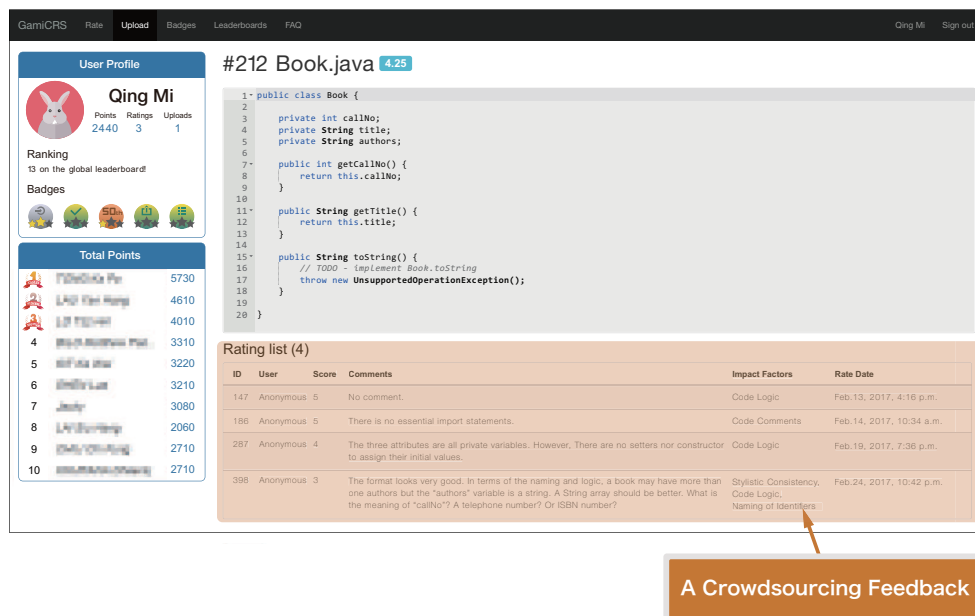


Figure 3.4: After uploading a code snippet, users can receive a crowdsourcing feedback regarding its readability level.

Note that GamiCRS allows only Java code for the following reasons: 1) The majority of previous studies build their code readability models based on Java programs (including but not limited to [17, 42, 99]). We keep consistent with their choice for comparison purposes. 2) To explore the feasibility of GamiCRS, we carefully plan a field experiment in a course held by the City University of Hong Kong. Java is one of the programming languages used in the course.

3.3.2 Development Framework

GamiCRS is implemented as a web-based application that can be accessed anywhere, any-time for the convenience of users. In order to make the web development fast and easy, we

adopt Django¹ as back-end framework to handle data storage and processing, and Bootstrap² as front-end framework to ensure a user-friendly interface.

Django is a free and open source Python framework that emphasizes the reusability and pluggability of components. By abstracting a set of common web development patterns (e.g., content administration and user authentication), Django reduces the workload of repetitive design, development, and deployment of complex websites.

Bootstrap is a popular front-end framework for designing websites and web applications, which contains dozens of simple and flexible HTML- and CSS-based UI components and layouts (e.g., typography, buttons, and navigation), as well as optional JavaScript extensions.

3.3.3 Code Snippet Repository

As shown in Figure 3.3, the entire code snippet repository is composed of two parts: an initial repository prepared by researchers before GamiCRS being put into use, and a subsequent repository generated by users during the running process of GamiCRS. In this section, we mainly describe the preparatory work of the initial code snippet repository. After that, we introduce the rules and strategies of how to select the next code snippet for users to rate.

3.3.3.1 Preparation of the Initial Code Snippet Repository

Prior to putting GamiCRS into service, we prepare a repository involving three datasets (Table 3.3), namely D_{Buse} , D_{Dorn} , and $D_{Scalabrino}$, which comprises a set of code snippets for which the readability degree has been evaluated by previous studies [17, 42, 99] using

¹<https://www.djangoproject.com>

²<https://www.getbootstrap.com>

a five-point Likert scale ranging from 1 (Very Unreadable) to 5 (Very Readable). The repository serves two purposes: to support the evaluation of GamiCRS (the details are shown in Section 3.4.2), and to guarantee the normal operation of GamiCRS before any user-uploaded code snippet appears.

Table 3.3: Statistical summary of the initial code snippet repository.

Dataset	# of Code Snippets ^a	# of Annotators	Programming Language	Lines of Code
D_{Buse}	100 $\rightarrow$ 50	120	Java	7.7 lines on average
D_{Dorn}	360 $\rightarrow$ 60	5468	Java, Python, and CUDA	roughly 10, 30, and 50 lines
$D_{Scalabrino}$	200 $\rightarrow$ 100	9	Java	10 to 50 lines

^aIn each field, the value to the left of the arrow is the total number of code snippets, while the value to the right is the number of selected ones.

D_{Buse} (Dataset Provided by Buse et al. [17])

Buse et al. claimed that the notion of readability is largely independent of the traditional measures of code complexity. They thus deliberately chose short code snippets with precisely three consecutive simple statements, in an attempt to eliminate context and mask complexity to a large extent. Figure 3.5a serves as a typical example.

D_{Dorn} (Dataset Provided by Dorn et al. [42])

To simulate the user experience of viewing a window of an arbitrary portion of code, Dorn et al. randomly extracted samples of different lengths (roughly 10, 30, and 50 LOC each) from all available source files. Similar to D_{Buse} , code snippets in D_{Dorn} could begin or end in the middle of a block or multi-line statement, as shown in Figure 3.5b.

$D_{Scalabrino}$ (Dataset Provided by Scalabrino et al. [99])

Noticing that the majority of code snippets in D_{Dorn} and D_{Buse} are only fragments representing a partial implementation, Scalabrino et al. focused on complete code entities (i.e., methods). An example is shown in Figure 3.5c.

In D_{Buse} , D_{Dorn} , and $D_{Scalabrino}$, each code snippet is accompanied by an average score representing its readability level. To determine whether a certain code snippet is readable

```

1 | Node r = x.getRight();
2 |
3 |     if (r != null) {
4 |         x = r;
5 |
6 |         Node l = x.getLeft();

```

(a) Sample code snippet from D_{Buse} .

```

1 |         try {
2 |             if (selenium.isVisible("//section")) {
3 |                 break;
4 |             }
5 |
6 |             catch (Exception e) {
7 |             }
8 |
9 |             Thread.sleep(1000);
10 |         }

```

(b) Sample code snippet from D_{Dorn} .

```

1 | protected Object[] getUmlActions() {
2 |     Object[] actions = {
3 |         getActionPackage(),
4 |         getActionClass(),
5 |         null,
6 |         getAssociationActions(),
7 |         getAggregationActions(),
8 |         getCompositionActions(),
9 |         getActionAssociationEnd(),
10 |        getActionGeneralization(),
11 |        null,
12 |        getActionInterface(),
13 |        getActionRealization(),
14 |        null,
15 |        getDependencyActions(),
16 |        null,
17 |        getActionAttribute(),
18 |        getActionOperation(),
19 |        getActionAssociationClass(),
20 |        null,
21 |        getDataTypeActions(),
22 |    };
23 |
24 |     return actions;
25 | }

```

(c) Sample code snippet from $D_{Scalabrino}$.

Figure 3.5: Samples from the initial code snippet repository.

or not, a cut-off value should be specified. Code snippets with readability scores greater than the specified value are labeled as *Readable*, while the rest are labeled as *Unreadable*. Unsurprisingly, the cut-off value varies in different studies (e.g., 3.14 in D_{Buse} and 3.6 in $D_{Scalabrino}$). We thus remove code snippets with neutral or ambiguous judgments, and keep only the top 25% and the bottom 25% samples according to their readability scores. The absolute number is 50, 60, and 100 for D_{Buse} , D_{Dorn} ³, and $D_{Scalabrino}$ respectively, as shown in Table 3.3.

3.3.3.2 Selection Strategy for Next Code Snippet

As described above, we have well prepared a fairly large repository for GamiCRS. Next, we need to set priorities for which code snippet should be displayed first for users to evaluate. In our original design, only two simple rules have been set up:

³Note that D_{Dorn} is composed of code snippets taken from Java, Python, and CUDA projects. This study concerns only the Java samples.

- Users cannot rate self-uploaded code snippets.
- Users cannot rate code snippets they have been rated before.

To take full advantage of the gamification mechanisms, we bring forward two additional rules as the balanced trade-off of encouraging high points users and allowing every code snippet to receive as many ratings:

- Code snippets uploaded by users with high points are prioritized.
- Code snippets with few ratings are prioritized.

According to these criteria, we suggest the following selection procedure:

- Randomly select M code snippets uploaded by users with relatively high points.
- Randomly select N code snippets with relatively few ratings.
- Make the final selection randomly from the $M + N$ candidates, where M and N are customizable.

3.3.4 Rules and Regulations of PBL

Among all relevant game design elements (GDEs), we adopt the classic PBL triad (Points, Badges, and Leaderboards) in GamiCRS. In this section, we systematically describe the rules and regulations specific to each PBL.

Our design process begins by identifying the core activity in GamiCRS. Consistent with traditional code readability surveys, GamiCRS aims to obtain as many high-quality readability ratings from human experts as possible. Thus, either directly or indirectly, all our designs center around this purpose. We expect that the competitive nature of PBL is capable of encouraging users to do their best to accomplish the designated task.

3.3.4.1 Point System

Points are used to reward users' behaviors. Each user is shown his/her current points calculated according to Table 3.4. It should be noticed that not all behaviors are created equal. We reward more points for those behaviors requiring more effort and attention, or can be more beneficial to the survey completion.

Table 3.4: Point system in GamiCRS.

Activity	Point	Description
Participate on GamiCRS	1000	One time only
Log in to GamiCRS	10+	Once a day More points for consecutive login ^a
Rate a code snippet	50	Extra 20 points for comments
Upload a code snippet	20	Up to 1000 points per day
Obtain badges	50+	Detailed information in Table 3.5
Maintain a position on points leaderboard	100+	Once a week For top 10 users only ^b

^a10 points for 1 day login, 20 points for 2 days login, etc.

^b1000, 600, 400 points for top 3 users, 100 points for others.

Considering that users may focus on repetition of non-useful gaming behaviors to get plenty of points at one stroke (e.g., rate as many code snippets as they can, regardless of the quality), we set a ceiling on the maximum points users can be given per day to discourage such meaningless behaviors. As shown in Table 3.4, GamiCRS allows a maximum of 50 ratings or uploads per day. Users will be informed once they reach this predetermined limit.

3.3.4.2 Badge System

Badges are essentially virtual symbols of accomplishments. To earn a certain *Badge*, users need to complete the *Activity* that goes with the *Badge*. Table 3.5 enumerates all the 20 badges available in GamiCRS, which can be further categorized into 4 groups (i.e., basic,

standard, premium, and professional) according to the difficulty of accomplishing the *Activity*. In particular, basic ones take the role of instruction to show what's possible in the system as they are extremely easy to get, while the others are used to provide various goals for users to strive towards.

Table 3.5: Badge system in GamiCRS.

Category ^a	Badge	Activity
Basic	New Comer	The first time login
	Promoter	The first time rating a code snippet
	Talkative	The first time commenting on a code snippet
	Enlightened	The first time uploading a code snippet
	Competitor	The first time getting a ranking
Standard	Enthusiast	For login on 3 separate days
	Explainer	Rate at least 20 code snippets
	Commentator	Comment on at least 10 code snippets
	Junior Engineer	Upload at least 10 code snippets
	Mortarboard	Enter the top 50 in points leaderboard
Premium	Activist	For login on 7 separate days
	Refiner	Rate at least 100 code snippets
	Pundit	Comment on at least 50 code snippets
	Senior Engineer	Upload at least 50 code snippets
	Epic	Enter the top 10 in points leaderboard
Professional	Fanatic	For login on 14 separate days
	Illuminator	Rate at least 200 code snippets
	Altruist	Comment on at least 100 code snippets
	Principal Engineer	Upload at least 100 code snippets
	Legendary	Win the first place in points leaderboard

^a50, 200, 300, and 500 points are awarded to users for obtaining basic, standard, premium, and professional badges respectively.

3.3.4.3 Leaderboard System

Leaderboards can be used to visually reflect a user's current position in regards to his/her peers, which is a great way to fuel friendly competition. However, if the gap between the

user and the champion is too great, he/she may feel frustrated and lose motivation towards the activity. One way to prevent this scenario is to simply involve multiple leaderboards across a variety of contexts and time frames, in an attempt to demonstrate users' endeavors and achievements from different perspectives. Thus, we carefully introduce 5 long-term (i.e., *Total Points*, *Total Ratings*, *Total Uploads*, *Total Badges*, and *Total Login Days*) and 3 short-term (i.e., *Daily Points*, *Daily Ratings*, and *Daily Uploads*) leaderboards in GamiCRS. Note that all daily leaderboards are reset at midnight every day.

3.4 A Field Experiment on GamiCRS

We assume that the incorporation of gamification techniques and infrastructure into the context of code readability survey can produce psychological (e.g., satisfaction and fun) and behavioral (e.g., participation and performance) benefits for survey participants. To scientifically test this hypothesis, a field experiment is conducted in a real-life environment to compare GamiCRS with its non-gamified counterpart, which lasts over a period of 2 weeks involving a class of 161 undergraduate students from the City University of Hong Kong.

3.4.1 Experimental Design

3.4.1.1 Participants

The experiment is performed in a large undergraduate course *CS3342 Software Design*, held by the City University of Hong Kong, which aims to introduce the fundamental principles and practice of software process and object-oriented software development methodologies. We have two primary reasons for choosing CS3342 in this research. First, a total of 161 students has enrolled in the course. The class size is large enough to well support our

experiment. Second, since having familiarity with Java programming language (or equivalent) is the prerequisite for enrolling in CS3342, we conjecture that all students have some programming experience, which is necessary for our experiment.

To avoid the carryover effects, and lower the chances of participants suffering fatigue or boredom from a long experimental duration, we carefully plan a between-subjects design [22]. All participants are randomly and evenly allocated to the treatment and control group. After successfully logging into GamiCRS with the pre-allocated *Username* and *Password*, participants are automatically directed to different interfaces according to the group they belong to. The treatment group ($N = 81$) has various GDEs displaying in the user interface to encourage their contributions, while the control group ($N = 80$) does not. All other respects are treated identically to eliminate the impact of environmental variables.

3.4.1.2 Procedures

The experiment spans a period of 14 days from 13th February to 27th February. To reflect the natural behaviors and avoid creating time pressure, participants are permitted to access GamiCRS at their leisure. The specific process of the experiment is described as follows:

- **13th February:** Participants are told that they are invited to attend a code readability survey and they are free to withdraw from the experiment at any time they want. All they need to do is rate Java code snippets on the readability based on their knowledge and experience, and their participation would assist in a study of identifying factors that affect code readability. After that, a simple demonstration of the basic operation of GamiCRS is provided.
- **20th February:** Participants are reminded that the experiment is still in progress, and their support and cooperation would be greatly appreciated.

- **27th February:** A post-experiment questionnaire is given to participants to collect demographic information (e.g., age and gender) as well as their perspectives on GamiCRS.

Note that the experimental procedures and main tasks are introduced to participants, but not the true objective, this is to protect our experiment from the observer-expectancy effect [55].

3.4.2 Experimental Results

In this section, we report to what extent the inclusion of gamification within code readability survey has an impact on participants' performance in terms of productivity and quality.

At the end of the field experiment, we obtain a set of subjective readability ratings. The number is 385 versus 355 for the group of treatment and control respectively. For the productivity aspect, the treatment group produces slightly more ratings as compared to the control group, yet it appears to be no significant difference between the two comparison groups.

Prior to assessing the quality, we consider the inter-rater agreement to evaluate whether it is possible to aggregate all the subjective readability ratings into a coherent model. By referring to previous studies [17, 42], we employ Spearman's rank correlation coefficient (Spearman's rho) to measure the agreement degree among independent raters. Two specific reasons are provided to support our choice: 1) The readability scores are essentially ordinal scales of non-numeric concepts. It is the order that matters rather than the magnitude. Specifically, if both raters assign code snippet X a higher readability score than code snippet Y, we consider that they share a consistent view regarding these two snippets, irrespective of the absolute numerical values [17]. 2) It is not reasonable to assume that the readability

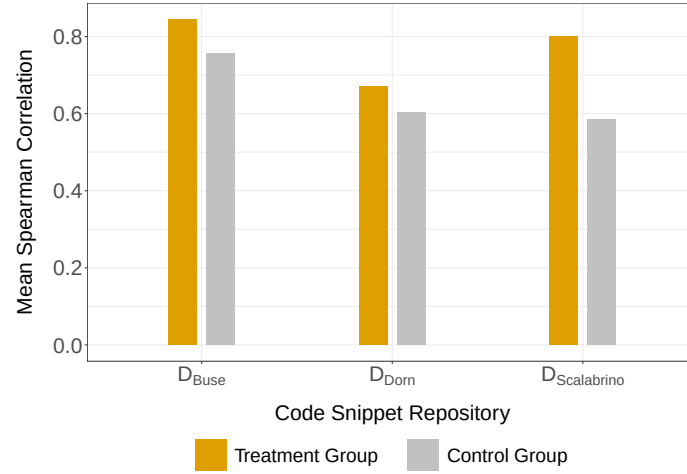
scores follow a normal distribution. Thus, we choose the nonparametric Spearman's rho instead of the Pearson product-moment correlation coefficient.

To obtain the inter-rater agreement, we first specify the ground-truth in terms of readability for each code snippet. Based on the specified ground-truth, we compute Spearman's rho for each rater and then average these correlations. The result can range in value from -1 to 1. Note that the raters who provide only one or two ratings have been excluded, as Spearman's rho is meaningless in such a situation.

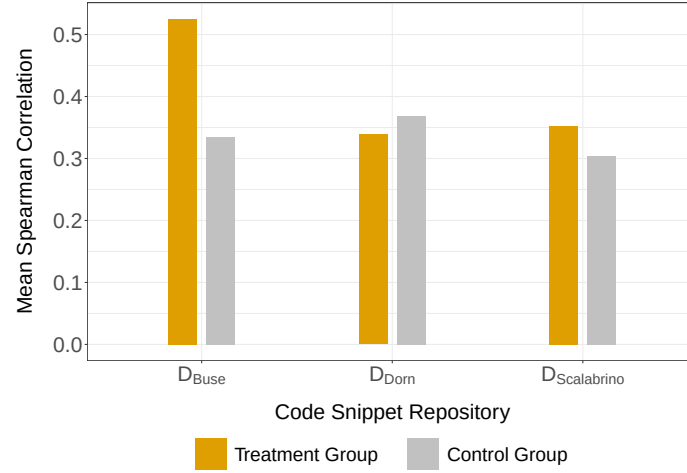
Figure 3.6a displays the inter-rater agreement with the average rating scores to be the ground-truth, which is calculated and compared for D_{Buse} , D_{Dorn} , and $D_{Scalabrino}$ separately due to the fundamentally different characteristics of these datasets (as shown in Figure 3.5 and Table 3.3). Our interpretation for the value of mean Spearman correlation is in line with those from Buse et al.'s work [17]. They claimed that for a human-related study, a correlation greater than 0.5 is typically considered as moderate to strong. It can be observed that the results for both treatment and control groups are above the prescribed limit, which validates the effectiveness of our unified model. Also, it is important to note that the treatment group demonstrates a higher inter-rater agreement across all datasets as compared to the control group.

So far, each code snippet is accompanied by two scores: a standard score comes from previous studies [17, 42, 99], which we consider as the correct answer, and a mean-opinion-score that has been validated by the inter-rater agreement analysis, which represents the participants' judgments during the field experiment. Next, we evaluate the quality of the mean-opinion-score.

As is described in Section 3.3.3.1, each dataset can be divided into two groups based upon the standard score, namely *Readable Group* (the top 25% code snippets with a high



(a) Mean Spearman correlation with the average readability judgments as the ground-truth.



(b) Mean Spearman correlation with the standard readability judgments as the ground-truth.

Figure 3.6: Comparison of inter-rater reliability between the *Treatment* and *Control* group.

readability level) and *Unreadable Group* (the bottom 25% code snippets with a low readability level). Using the nonparametric Wilcoxon-Mann-Whitney test [70] at 0.01 significance level, we examine whether the mean-opinion-scores in *Readable Group* are statistically significantly higher than those in *Unreadable Group*. To assess the magnitude of the difference between the two comparison groups, we calculate Cliff's Delta [24].

Table 3.6 illustrates the statistical results of Wilcoxon-Mann-Whitney test (*p-value* column) and the corresponding effect size (*d-value* column) for each dataset. The magnitude of Cliff’s Delta is interpreted using the thresholds provided in Romano et al.’s study [92], which is considered *Negligible* for $|d\text{-value}| < 0.147$, *Small* for $|d\text{-value}| < 0.330$, *Medium* for $|d\text{-value}| < 0.474$, and *Large* for otherwise.

Table 3.6: Comparison of mean-opinion-scores between the *Readable* and *Unreadable* group.

Dataset	Treatment Group		Control Group	
	<i>p-value</i>	<i>d-value</i> ^a	<i>p-value</i>	<i>d-value</i> ^a
D_{Buse}	0.001	0.520 (Large)	0.082	0.283 (Small)
D_{Dorn}	0.215	0.1833 (Small)	0.002	0.467 (Medium)
$D_{Scalabrino}$	0.000	0.412 (Medium)	0.062	0.214 (Small)

A significant difference with at least a small effect size (i.e., *p-value* < 0.01 and $|d\text{-value}| > 0.147$) is highlighted in Table 3.6. It can be observed that participants in the treatment group perform much better than the control group with regard to D_{Buse} and $D_{Scalabrino}$, but not D_{Dorn} . To verify our conclusion, we take the standard readability judgments as the ground-truth to calculate the mean Spearman correlation one more time. The result is presented in Figure 3.6b, which reaffirms that participants agree with the correct answer more often in the treatment group than do the control group.

3.4.3 Post-Experiment Questionnaire

After the field experiment, a questionnaire approach is utilized in the treatment group to investigate users’ impressions and attitudes toward GamiCRS based on Technology Acceptance Model (TAM) [30, 31], which is one of the most robust theories for predicting and/or explaining user acceptance. This research is among the first to employ TAM in the field

of gamified survey. Specifically, we structure our post-experiment questionnaire with four TAM-based Likert-subscales modified to the context of this study.

- **Perceived Ease of Use (PEOU)**: the extent to which users believe that using GamiCRS are easy (i.e., free of physical and mental effort).
- **Perceived Usefulness (PU)**: the extent to which users believe that using GamiCRS can enhance their performances and help achieve their goals.
- **Attitude Toward Game Design Elements (ATGDE)**: an overall assessment representing users' attitudes toward the effects of game design elements.
- **User Satisfaction (US)**: the extent to which users consider GamiCRS meets their needs and expectations.

The objective of the post-experiment questionnaire is basically to understand users' perceptions and behaviors on GamiCRS. To fully capture the concept being assessed, each subscale is composed of a series of Likert-type items that represent similar statements, which is enumerated in Figure 3.7.⁴ Participants are required to specify their level of agreement to each statement on a scale ranging from 1 (Strongly Disagree) to 5 (Strongly Agree).

A total of 73 valid responses is received through both online and written questionnaires, giving a response rate of around 90% ($N = 81$). We begin by summarizing the demographic characteristics of the participants. The population comprises 54 (74%) males and 19 (26%) females. Approximately 89% of the participants report to have Java development experience (Median = 1, Mean = 1.56, SD = 1.44). As all participants are undergraduate students, their age varies in a small range from 18 to 26 years old (Median = 20, Mean = 20.59, SD = 1.49).

⁴The full version of the post-experiment questionnaire is provided in Appendix D.

Next, Cronbach's alpha (a measure of internal consistency or reliability) [27] is used to provide evidence that the Likert-type items form an internally reliable subscale (i.e., the Likert-type items are homogeneous and measuring the same concept). The value of Cronbach's Alpha for each of the subscale is shown in Table 3.7. Among a wide variety of standards proposed to interpret what makes a *good* alpha coefficient, we follow the rules of thumb as given in DeVellis et al.'s work [38], which is considered *Excellent* for >0.90 , *Very Good* for >0.80 , *Respectable* for >0.70 , *Minimally Acceptable* for >0.65 , *Undesirable* for >0.60 , and *Unacceptable* for <0.60 . It can be observed that all of our results are far above the threshold 0.6, thus demonstrating a reliable item-scale correlation coefficient.

Table 3.7: Internal consistency of the post-experiment questionnaire.

Subscale	Cronbach's Alpha	95% Confidence Boundaries		Mean	SD
		Lower	Upper		
PEOU	0.75	0.66	0.85	3.3	0.59
PU	0.85	0.79	0.91	3.5	0.71
ATGDE	0.90	0.87	0.94	3.2	0.73
US	0.86	0.80	0.91	3.1	0.78

After that, we provide the response distribution of the post-experiment questionnaire in Figure 3.7. The stacked bar chart is the most widely used approach for visualizing Likert results, with a position further to right indicating a higher level of agreement on the particular statement.

3.4.4 Implications

In this section, we provide implications of the experimental results for those who plan to start a research on gamification applications.

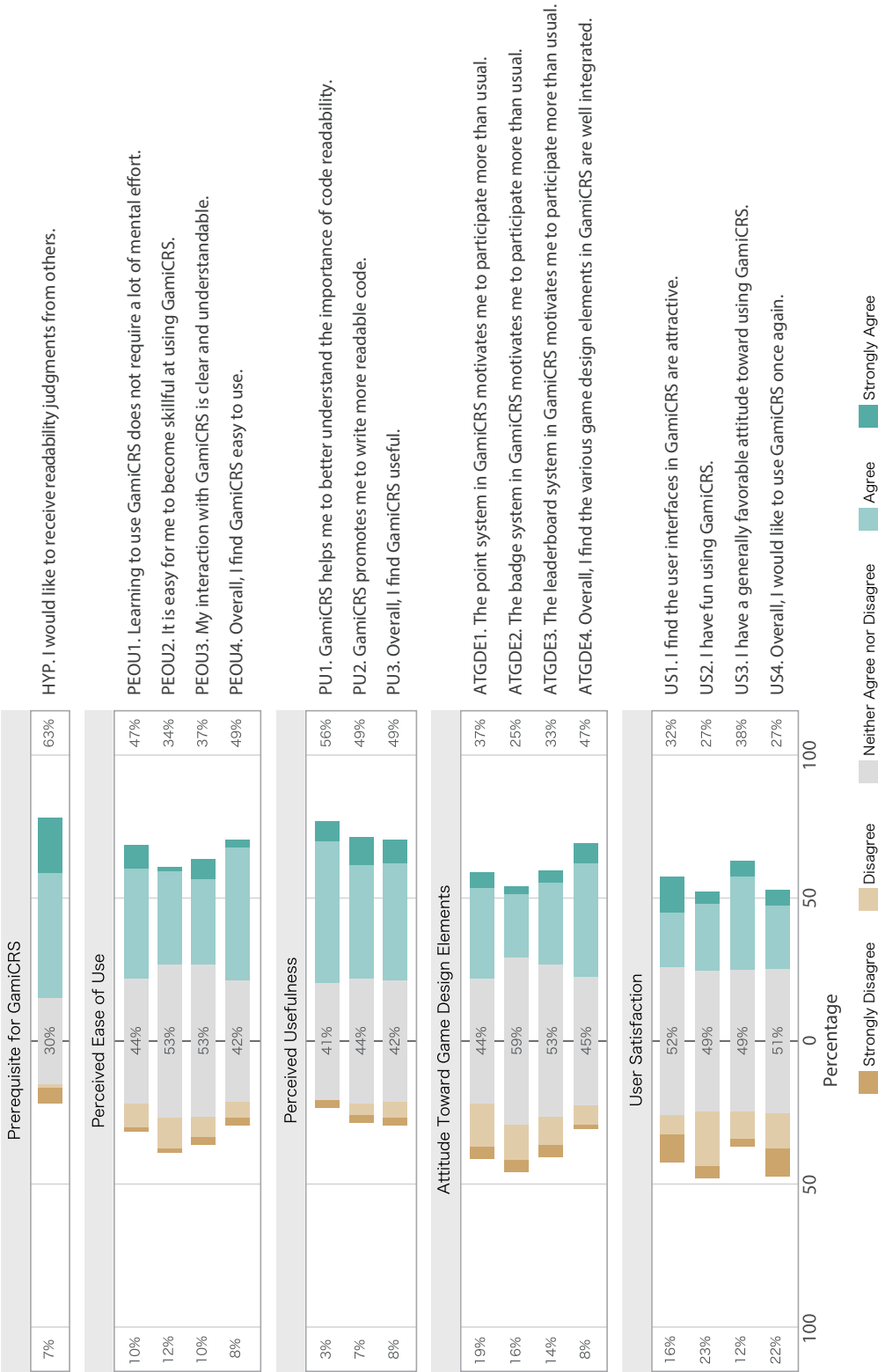


Figure 3.7: Responses to the post-experiment questionnaire.

GamiCRS has been proposed to make code readability survey more engaging and rewarding, with the underlying goal of improving the quality of survey results. A field experiment has validated the potential of GamiCRS on the performance of survey participants. However, it is noteworthy that the amount of collected data for both groups are low, which confirms the necessity of involving more persuasive factors (e.g., reciprocity, authority, scarcity, and brevity besides gamification [103]).

Since users' needs and desires should be put at the center of any gamification project, we value their feedback from the post-experiment questionnaire as an important measurement. The statement HYP in Figure 3.7 indicates that the majority of the participants would like to receive readability judgments from others, with more than 60% of responses in agreement and just 7% in disagreement, reflecting a very positive belief. When asked if they find using GamiCRS is easy and useful, most of the participants express a positive stance, which makes it possible to promote long-term behavioral changes with the help of GamiCRS. After that, we identify the impact of isolated GDEs to help researchers and practitioners to prioritize which GDEs to use in their future work. We discover that points and leaderboards are more effective than badges in motivating users to contribute. Although the PBL triad plays an active role, participants put more emphases on the usefulness of the gamified system, which can serve to guide future working directions. Also, participants express a slightly more neutral response concerning the US aspect, indicating that GamiCRS is not as attractive as what they expect. To improve GamiCRS, it would be valuable to conduct trials with different mechanisms, such as the inclusion of challenges (e.g., a non-deterministic rewarding mechanism) and/or explicit collaborations.

3.5 Discussion

In this section, we discuss several issues that need to be clarified and the potential application directions of GamiCRS.

3.5.1 Gamification Versus Pointsification

Gamification is a relatively recent concept and in many ways it can be confused with pointsification, a term coined by game designer Margaret Robertson [6, 100].

Pointsification typically refers to adding nothing more than a scoring system to a non-game environment and representing it as the core of the experience. Although pointsification is somewhat effective in motivating certain activities, it is usually questionable as it focuses solely on the least interesting and essential part of games - the scoring system. The caveat is that relying only on the valueless points as incentive mechanism is often insufficient, and on the other hand, could easily undermine users' inherent interest to pursue the activity. Gamification, by contrast, covers a much wider range of possibilities. Rather than just sticking together a collection of GDEs, meaningful gamification advocates systematic and artistic design for the purpose of fostering real engagement.

During the design process of GamiCRS, we deliberately adopt the strengths of pointsification and gamification while overcoming their shortcomings. Considering that the goal of GamiCRS is to create a long-term change where users feel positive about engaging in the survey process, we attempt the combination of both intrinsic and extrinsic motivators to form a systemic incentive mechanism. We begin by inspiring users' extrinsic motivation through various GDEs. At this stage, all users' endeavors and accomplishments are meticulously tracked, measured and therefore rewarded with points and badges. While keeping them in the reward loop, we gradually facilitate users' understanding of the importance of

code readability as the survey progresses, and ultimately internalize its regulation so they can be self-motivated and self-directed to perform the activity, instead of trying to escape it or just get through it.

3.5.2 Existing Problems and Challenges in GamiCRS

With the approach of bringing game mechanics into code readability survey, we develop GamiCRS. Despite our best efforts, there are still many problems and challenges that need to be resolved. In this section, we list out all the issues being found during the course of the field experiment, in order that researchers and practitioners who are interested in the relevant topic could avoid committing the same mistakes.

- One major concern is the repetition of gaming behaviors. Striving for a high score, some participants perform a considerable number of ratings/uploads within a short period of time, which inevitably raises doubts as to the data quality. A more effective strategy should be developed in future work to restrict these meaningless behaviors.
- Contrary to our expectations, only a small amount of user-uploaded code snippets appear in GamiCRS. A possible reason is that the only link pointing to upload page is placed in an inconspicuous position. Moreover, as we expect participants to discover the upload feature through their own efforts, we have not mentioned or demonstrated it prior to the initiation of the field experiment, which seems inappropriate in this context.
- In current GamiCRS, users can receive only a modeless messagebox displaying the given points after rating a code snippet. A student reports that he/she prefers a more useful and relevant feedback, for instance, whether or not his/her judgment is correct, or conforms to previous judgments provided by others.

- Another problem, pointed out by a student, is that the code snippet in GamiCRS is rendered with syntax highlighting in a form different from modern IDEs (e.g., Eclipse⁵), which may well lead to users' confusion and frustration.

The first two issues are identified through the experimental results, while the last two through the open question “Any other comments and suggestions?” at the end of the post-experiment questionnaire. We encourage researchers and practitioners to give serious consideration to these advices in their future work.

3.5.3 Threats to Validity

We discuss potential threats to the validity of our study in this section.

3.5.3.1 Internal Validity

Several factors limit the findings of this study. First, among various GDEs, we adopt the well-known PBL triad (Points, Badges, and Leaderboards) to implement game experiences, as they are simple yet proved to be effective in most previous gamification projects. However, it is highly possible that a different set of GDEs (or a different set of rules and regulations regarding GDEs) may yield different experimental results. Further research is necessary to investigate the effects that different (combinations of) GDEs may have on GamiCRS.

Another possible limitation to this study is the duration of the field experiment. All participants spend only a short period of time (just 2 weeks) using GamiCRS, which may not be considered enough to derive a general conclusion. On the other hand, the long-term effects of GamiCRS remain to be further observed due to the limited time-span. This

⁵<https://www.eclipse.org>

threat to generalization could be mitigated by extending the experimental period to provide a comprehensive analysis in future research.

3.5.3.2 External Validity

To ensure the effectiveness of GamiCRS, we carefully plan a field experiment with 161 participants. We believe that the sample size is large enough to validate our results. However, all participants involved in the experiment are undergraduate students at the City University of Hong Kong, sharing a relatively homogeneous background in terms of age and technology skills, which may not be representative of the general community. Although the use of college students imposes some limits on our external validity, it is not believed that the overall results are skewed, yet further study is required to extrapolate our findings to other populations.

3.5.3.3 Construct Validity

Another consideration resides in the post-experiment questionnaire designed on the basis of Technology Acceptance Model to capture users' impressions and attitudes toward the acceptance of GamiCRS. Although we have referred to a wide variety of literature to guarantee the quality and reliability of the questionnaire, by no means do we claim that it is capable of completely reflecting the concept being assessed. In addition, our choice of some Likert-type items are admittedly subjective. For future work, we intend to refine the questionnaire by introducing a wider range of fine-grained items to mitigate this threat.

Chapter 4

DeepCRM: A Deep Learning-Based Code Readability Model

Code readability classification has attracted increasing concern in academia and industry. To construct accurate classification models, previous studies (including but not limited to [17, 89]) usually handcrafted surface-level features such as operator counts or line lengths [42] to represent code readability. The process is admittedly labor-intensive, and it is possible that some important factors will be missed. Thus, instead of manual design and extraction of features, we introduce a deep learning-based approach that can learn complicated underlying features automatically from the source code.

In particular, we use Convolutional Neural Networks (ConvNets). We first introduce a representation strategy (with different granularities) to transform source codes into integer matrices as the input to ConvNets. We then propose DeepCRM, a deep learning-based model for code readability classification. DeepCRM consists of three separate ConvNets with identical architectures that are trained on data preprocessed in different ways. The

approach is not another model based on feature engineering, but a constructive method of gaining a deep understanding of the input data, which makes our study novel in this regard.

We evaluate our approach against five state-of-the-art code readability models. The experimental results show that DeepCRM can outperform previous approaches. The improvement in accuracy ranges from 2.4% to 17.2%. By eliminating the need for manual feature engineering, DeepCRM provides a relatively improved performance, confirming the efficacy of deep learning techniques in the task of code readability classification.

The rest of this chapter is organized as follows. Section 4.1 introduces some preliminaries. In Section 4.2, we describe the proposed approach. Section 4.3 presents the detailed design of our experiments and Section 4.4 summarizes the results. In Section 4.5, we present several exploratory experiments. Section 4.6 discusses our findings and the possible threats to validity.

4.1 Preliminaries

In this section, we introduce the basic knowledge regarding Convolutional Neural Networks (ConvNets) and the statistical methods used in this study.

4.1.1 Convolutional Neural Networks

Convolutional Neural Networks (ConvNets) [21, 65] are special kinds of Neural Networks with biologically inspired structures. As shown in Figure 4.1, a typical ConvNet is composed of a succession of convolutional, pooling, and fully-connected layers:¹

- **Convolutional Layer**

¹The ConvNet in Figure 4.1 is actually LeNet-5 [107, 64], which is designed for recognition of handwritten and machine-printed characters.

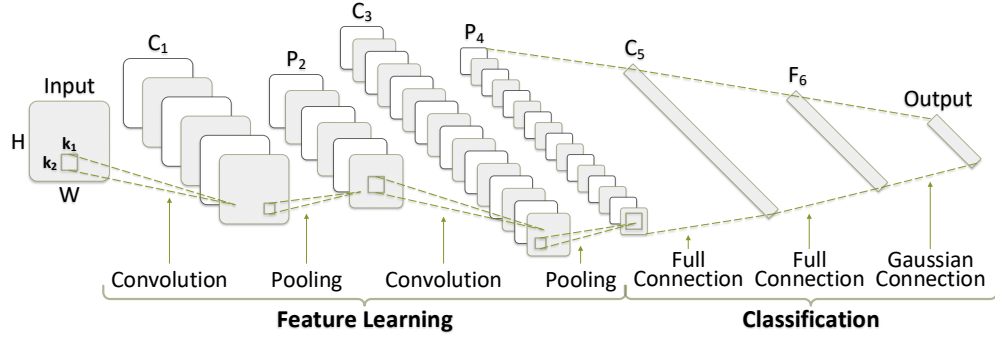


Figure 4.1: Architecture of a Typical ConvNet

A convolution is often interpreted as a filter. By applying a convolution operation followed by a nonlinearity to the input, we can obtain a feature map (the plane in Figure 4.1). Specifically, the output y at l^{th} layer is given by:

$$x_{i,j}^l = \sum_m \sum_n w_{m,n}^l y_{i+m,j+n}^{l-1} + b^l \quad (4.1)$$

$$y_{i,j}^l = \sigma(x_{i,j}^l) \quad (4.2)$$

where input x is of dimension $W \times H$ (i.e., *width* $\times$ *height* of input x , see Figure 4.1) and has i by j as the iterators; filter w is of dimension $k_1 \times k_2$ (i.e., *width* $\times$ *height* of filter w , see Figure 4.1) and has m by n as the iterators; b^l denotes the bias unit at l^{th} layer; and $\sigma(\cdot)$ denotes a nonlinear activation function such as Sigmoid ($\frac{1}{1+e^{-x}}$), Tanh ($\frac{e^x - e^{-x}}{e^x + e^{-x}}$), or ReLU ($\max(0, x)$). The output feature maps serve as the input to the next layer.

• Pooling Layer

After a convolutional layer, a pooling layer is usually added to perform down-sampling by concentrating the input over a certain area into a single value. The process can help

relieve the computational load because it reduces the number of parameters in the network. Various techniques can be used in pooling layers, including max-pooling and mean-pooling.

- **Fully-Connected Layer**

Fully-connected layers are allocated after a couple of convolutional and pooling layers. Each neuron in fully-connected layers connects to all activations in the previous layer.

From another perspective, a typical ConvNet consists of a feature learning network (including one or more convolutional and pooling layer pairs) and a classification network (including one or more fully-connected layers) [57]. The former network extracts features automatically from the input data, whereas the latter network generates output based on the extracted features. The system parameters (i.e., weights w and biases b) are adjusted via the training process.

4.1.2 Statistical Methods

- **The Brunner-Munzel Test**

To detect differences between two comparison groups, we adopt the Brunner-Munzel test [16, 81] as recommended by Kitchenham et al. [59], which is a derivation of the Wilcoxon-Mann-Whitney test [70]. The Brunner-Munzel test has the advantage of making no assumptions regarding the shape or continuity of the underlying distribution, which works well (even for small and unbalanced sample sizes) compared to alternative methods [82].

- **The Cliff's Delta Statistic**

To determine the magnitude of the differences between the two comparison groups, we adopt the Cliff's Delta statistic [24] as recommended by Kitchenham et al. [59]. Cliff's

Delta is a measure of dominance, which estimates the probability that a randomly selected observation from one group is larger than a randomly selected observation from another group, minus the reverse probability [69].

Specifically, the Cliff’s Delta estimator is given by:

$$d = \frac{Sum[x > y] - Sum[x < y]}{N_x N_y} \quad (4.3)$$

where the two comparison groups x and y are of size N_x and N_y , respectively. We interpret the value of Cliff’s Delta using the thresholds provided in Romano et al.’s study [92], which is considered *Negligible* for $|d\text{-value}| < 0.147$, *Small* for $|d\text{-value}| < 0.330$, *Medium* for $|d\text{-value}| < 0.474$, and *Large* for otherwise.

4.2 Proposed Approach

Analogous to priorwork [17, 89, 42, 99], we regard the code readability classification problem as a binary classification problem: given a piece of source code $x =$

$$\begin{pmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mm} \end{pmatrix}, \text{ where } m \text{ denotes lines of code, } n \text{ denotes the maximum line length,}$$

and x_{ij} denotes a code symbol.² We classify x into a *Readable* or *Unreadable* class.

As shown in Figure 4.2, our approach consists of three major stages: 1) transformation of source codes into integer matrices as the input to ConvNets; 2) construction of multiple ConvNets and training with respect to learnable parameters; 3) determination of whether a new instance is readable or unreadable using the trained model. In this section, we describe the first two stages in detail.

²The code symbol acts as the basic unit of the source code, which can be a character, a keyword, a function, etc. Further details are provided in Section 4.2.1.

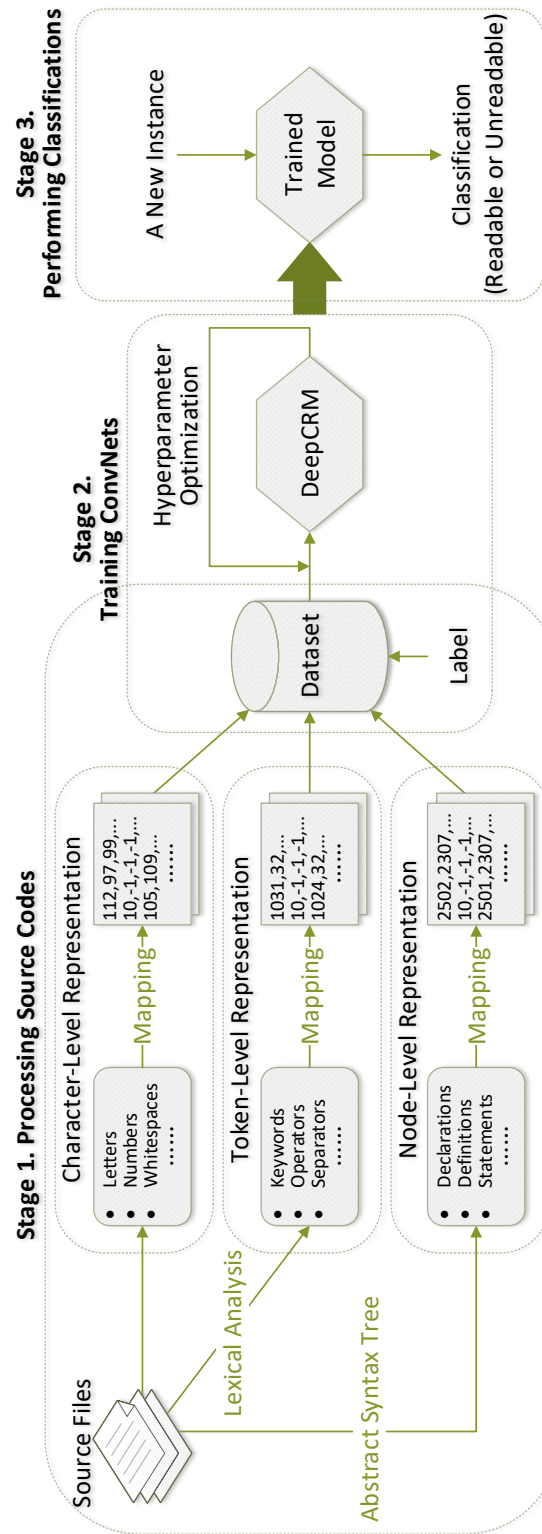


Figure 4.2: General Overview of the Proposed Approach

4.2.1 Source Code Representation

To enable deep learning-based program analyses, we must formulate an appropriate strategy for source code representation.

As shown in Figure 4.3, previous studies on deep learning applications in software engineering often preserve partial information of the source code. For instance, Wang et al. [109] made use of only three types of AST nodes (i.e., invocation nodes, declaration nodes, and control-flow nodes) for defect prediction. Gu et al. [49] focused solely on API usage sequences (e.g., *FileOutputStream.new*, *FileOutputStream.write*, and *FileOutputStream.close*) for API recommendation. Because there is no definitive answer as to which factor(s) can affect code readability, our basic principle is to preserve as much as possible the original information about the source code.

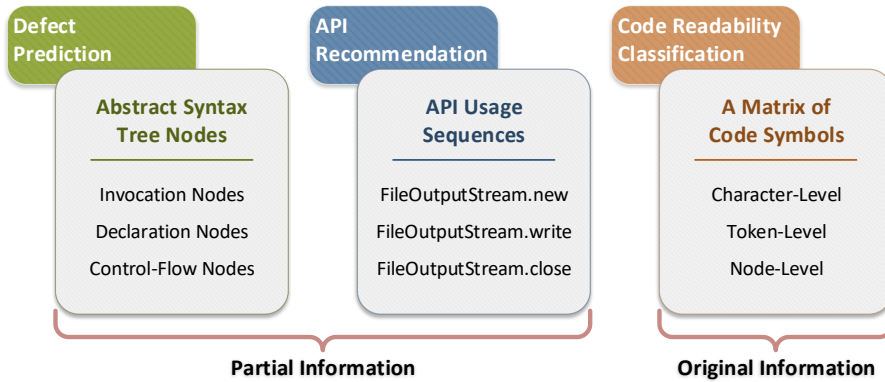


Figure 4.3: Different Strategies for Source Code Representation

Specifically, we first parse the source codes into a set of symbols and then convert the symbols into integer matrices, which is the universal format that ConvNets can easily analyze and manipulate. Possible granularities of the code symbol (x_{ij}) are enumerated as follows [29, 88]:³

³It is notable that we do not consider statement-, function-, or even higher-level representations, primarily because they are too abstract and may well lose useful information for code readability classification.

- **Character-Level Representation:** Each character (e.g., a-z, A-Z, and 0-9) is treated as a symbol.
- **Token-Level Representation:** Each token (e.g., keywords and operators) is treated as a symbol.
- **Node-Level Representation:** Each node in the Abstract Syntax Tree (e.g., declarations and definitions) is treated as a symbol.

In this section, we detail our representation strategy specific to each granularity. Note that we focus solely on Java language because most previous studies constructed their code readability models based on Java programs (including but not limited to [17, 89, 42, 99]). We remain consistent with their choice for comparison purposes.

4.2.1.1 Character-Level Representation

Like the pixels of an image, we treat a source code as a matrix of characters. Specifically, we transform letters (i.e., a-z and A-Z), numbers (i.e., 0-9), and marks (e.g., parentheses and braces) into their ASCII values. Given that the spatial layout of the source code may well affect its readability, we also preserve whitespaces (i.e., spaces, horizontal tabs, and line terminators) that have generally been ignored in prior deep learning studies.

The final output takes the form of two-dimensional arrays. A simplified example is presented in Figure 4.2. Note that we pad the arrays with a special integer (here, -1) because ConvNets require all inputs have the same length.

4.2.1.2 Token-Level Representation

Referring to the tokenization mechanism proposed by Basit et al. [7],⁴ we too assign a unique integer to each token, including keywords (e.g., `public` and `int`), separators (e.g., semicolons and brackets), operators (e.g., subtraction and decrement), literals (e.g., `true` and `null`), and whitespaces. The advantage of this method is its simplicity and customizability. For instance, if we consider the difference among data types *byte*, *short*, *int*, and *long* to be insignificant for code readability classification, we can specify an integer to uniformly represent them.

It is noteworthy that some tokens such as variable names are user-defined and thus method-specific, which cannot be generalized to other code snippets. If we similarly assign every such token with a unique integer, it is likely to cause the undesired data sparseness. One possible approach to handle this situation is to introduce a suitable abstraction, such as replacing all variable names with a particular value.⁵ Considering that useful information may be available in user-defined tokens (e.g., naming conventions [12, 101]), we put forward an alternative method that transforms these tokens with their ASCII values in an attempt to preserve as much information as possible.

4.2.1.3 Node-Level Representation

Programs usually contain well-defined syntax that can assist in detecting certain patterns [109, 83]. To capture the syntax information, we use the Abstract Syntax Tree (AST), which is an intermediate tree-like representation that allows us to access the syntactic structure of the source code. Specifically, we traverse each AST and encode every kind of node with a special integer. Note that ASTs omit certain elements in programs and thus enable

⁴The tokenization mechanism [7] is proposed for code clone detection. Because we are considering a different goal, we have made some necessary changes.

⁵A similar approach was adopted by Wang et al. [109].

the suppression of some inessential classes (e.g., the *byte*, *short*, *int*, and *long* examples discussed in the previous section). This is actually the most commonly used representation level in deep learning-based program analyses, including defect prediction [109] and code clone detection [112].

4.2.2 Neural Network Architecture

Corresponding to each granularity, we construct three separate ConvNets with identical architectures. For simplicity, we denote them as *ConvNet_{CR}* (CR stands for Character-Level Representation), *ConvNet_{TR}* (TR stands for Token-Level Representation), and *ConvNet_{NR}* (NR stands for Node-Level Representation), respectively. The objective is to have multiple ConvNets that are skillful, but from different perspectives. In each ConvNet, the feature learning network contains a convolutional layer with a bank of filters, followed by the ReLU function⁶ and a max-pooling layer. The classification network is composed of a fully-connected layer leading to a two-way Softmax classifier. The details are illustrated in Figure 4.4.

As we discussed in Chapter 1, code readability is essentially an intuitive concept. The best way to determine whether a source code is readable is to consult with a number of domain experts rather than depending on one individual’s judgment. To simulate this process, we propose the combination of multiple ConvNets into an ensemble model. The intuition is that the collective wisdom often produces a better solution [40]. Accordingly, we aggregate *ConvNet_{CR}*, *ConvNet_{TR}*, and *ConvNet_{NR}* with adjustable weights. We denote the resulting model as DeepCRM (Deep Learning-Based Code Readability Model). The main novelty of our approach is the inclusion of the ensemble architecture, which is shown in Figure 4.5.

⁶The ReLU function (Rectified Linear Unit [68]) is widely used in recent applications because it can yield comparable results but converge several times faster than their equivalents [60].

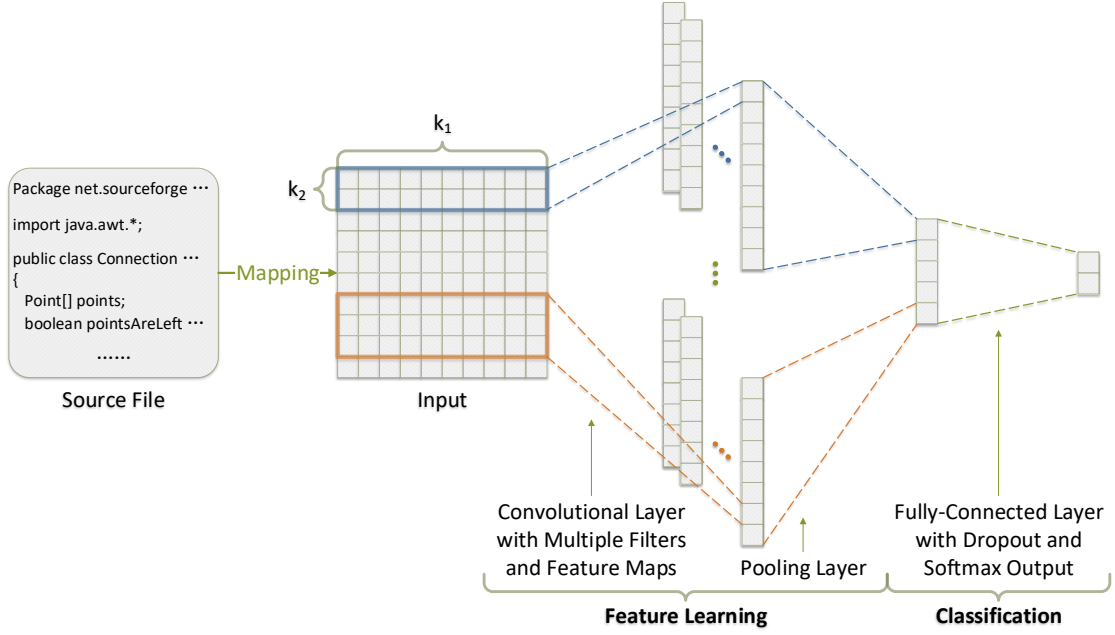


Figure 4.4: Architecture of Separate ConvNets

4.2.3 Details of Configuration and Training

The proposed ConvNets are implemented in Python using Keras⁷ and TensorFlow⁸. We train them in a supervised manner with the Adam optimization algorithm [58], which is highly recommended as the optimization method for deep learning applications [54]. The learning rate is initialized as 0.001 and adapted during training. The goal is to minimize the network's loss (or misclassification error) with respect to adjustable parameters (i.e., weights⁹ and biases). Here we adopt the widely used cross-entropy loss [32] for our classification problem:

$$L = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M t_{i,j} \log y_{i,j} \quad (4.4)$$

⁷<https://github.com/keras-team/keras>

⁸<https://www.tensorflow.org>

⁹Analogous to most ConvNets, we use a weight-sharing strategy to reduce the number of free parameters.

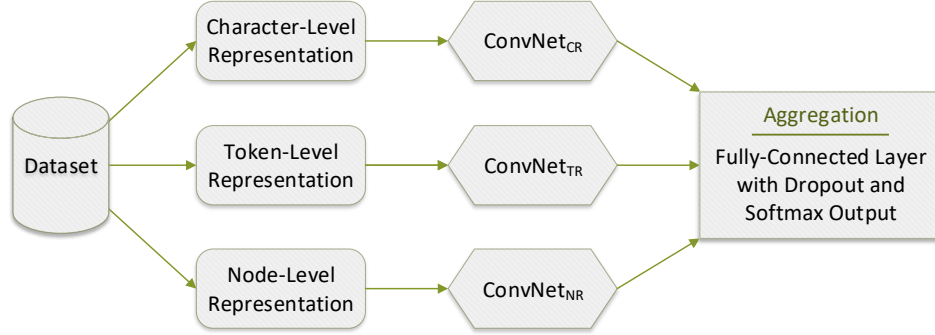


Figure 4.5: Ensemble Neural Network Architecture

where $N = 32$ is the number of training instances used in one iteration (i.e., the batch size), $M = 2$ is the number of classes (i.e., *Readable* and *Unreadable*), t is the ground truth, and y is the estimated class probability. The training data are randomly shuffled before they are sent to the network. To prevent over-fitting, a dropout with probability 0.5 is applied.

In addition to these configurations, we must fine-tune two hyperparameters: the filter size and the number of feature maps. The details are discussed in Section 4.4.

4.3 Experimental Setup

To validate the effectiveness and efficiency of our approach, we design several experiments. This section mainly describes the experimental setups. Specifically, we first outline five research questions and the corresponding motivations. We then detail the construction process of the dataset used in this study. Finally, the evaluation metrics are briefly introduced.

4.3.1 Research Questions

The goal of this study is to explore the potentiality of deep learning techniques in the task of code readability classification, with the purpose of improving the state-of-the-art. To evaluate whether deep learning is beneficial, we conduct a series of experimental evaluations. Specifically, we aim to answer the following research questions:

- **RQ1: Do different parameter settings affect the model performance?**

Motivation: We aim to explore the sensitivity of each ConvNet (*ConvNet_{CR}*, *ConvNet_{TR}*, and *ConvNet_{NR}*) with respect to different parameter settings. Because tuning is normally time-consuming, we expect our ConvNets to be relatively stable.

Approach: Two hyperparameters require fine-tuning: the filter size and the number of feature maps. To answer this research question, we vary the values of these two hyperparameters and evaluate their effects on the model performance in terms of accuracy.

- **RQ2: What is the benefit of the ensemble architecture?**

Motivation: To improve model generality, we propose a novel architecture that combines *ConvNet_{CR}*, *ConvNet_{TR}*, and *ConvNet_{NR}*. This research question involves investigation of the efficacy of this ensemble method.

Approach: We evaluate the performance of DeepCRM against the fine-tuned *ConvNet_{CR}*, *ConvNet_{TR}*, and *ConvNet_{NR}* using accuracy and f-measure.

- **RQ3: How accurate is DeepCRM in code readability classification when compared to traditional models?**

Motivation: This study is the first to apply ConvNets to the field of code readability classification. It is important to know whether our approach can outperform state-of-the-art code readability models.

Approach: To answer this research question, we compare DeepCRM with five competitors that are state-of-the-art models for code readability classification.

- **Buse et al.’s Model** [17]: Buse et al. proposed a set of local code features to represent code readability. They claimed that the model performed just as good as a human.
- **Posnett et al.’s Model** [89]: Based on Buse et al.’s work, Posnett et al. derived a model that is simpler, better performing, and theoretically well-founded with only 3 variables: $z = 8.87 - 0.033V + 0.40Lines - 1.5Entropy$.
- **Dorn et al.’s Model** [42]: Making use of visual, spatial, and linguistic features, Dorn et al. aimed to construct a universal model for code readability.
- **Scalabrino et al.’s Model** [99]: To improve the performance of the existing models, Scalabrino et al. further introduced a set of textual features targeted on identifiers and comments.
- **A Comprehensive Model** [99]: A code readability model that includes each of the features mentioned above.

We denote the aforementioned models as M_{Buse} , $M_{Posnett}$, M_{Dorn} , $M_{Scalabrino}$, and M_{All} successively. For a fair and easy comparison, these models are all trained using logistic regression as the underlying classifier [99].

• **RQ4: What is the time and space cost of training the proposed ConvNets?**

Motivation: Despite the advantages that deep learning may offer, its training process tends to be computationally expensive. In this research question, we aim to provide researchers and practitioners with a clear understanding.

Approach: We explicitly record the time and space cost of $ConvNet_{CR}$, $ConvNet_{TR}$, $ConvNet_{NR}$, and DeepCRM during the training process.

In summary, we begin by choosing the optimal hyperparameters for each ConvNet (RQ1). We then combine them into an ensemble model and evaluate its efficacy (RQ2). After that, we compare the performance of our approach with those of the state-of-the-art models (RQ3). Finally, we report the time and space cost incurred during the training process (RQ4).

4.3.2 Dataset Construction

To train and test our model, we first need to prepare a corpus of labeled code snippets as the ground-truth. Specifically, our data are extracted from prior code readability studies and open source software projects. In this section, we detail our data collection process.

To facilitate comparison with other approaches, we first gather a set of code snippets from previous studies, namely, D_{Buse} , D_{Dorn} , and $D_{Scalabrino}$, as shown in the first part of Table 4.1. In D_{Buse} , D_{Dorn} , and $D_{Scalabrino}$, each code snippet is evaluated by multiple human annotators on a five-point Likert scale [67] ranging from 1 (very unreadable) to 5 (very readable). We aggregate these subjective ratings by taking the mean to represent the readability degree of each code snippet. Following a similar approach to that of Lee et al. [66], we construct two representative groups for each dataset: the *Readable* group (i.e., the top 25% code snippets with high readability scores) and the *Unreadable* group (i.e., the bottom 25% code snippets with low readability scores), whereas the code snippets with neutral or ambiguous judgments (i.e., the middle 50% samples) are excluded.

Only a few human-annotated code snippets are available in the literature (see Table 4.1 for details), which may not be adequate to train the proposed ConvNets. We must extend our dataset with more samples. To achieve this, we first fully download ten Java projects

Table 4.1: Statistical Summary of the Dataset

Category	Dataset	Description	Source	# of Code Snippets		Lines of Code ^b	
				Total	Selected	Mean	SD
D_{CRS} (Labeled by Human Annotators)	D_{Buse}	Provided by Buse et al. [17]	SourceForge	100	50	7.80	2.47
	D_{Dorn}^a	Provided by Dorn et al. [42]	SourceForge	360	60	30.81	16.66
	$D_{Scalabrino}$	Provided by Scalabrino et al. [99]	SourceForge	200	100	26.61	10.38
D_{OSS} (Labeled by Automated Tools)	$D_{Hibernate}$	Object Relational Mapping Tool	Github	12254	6127	23.27	12.37
	$D_{HyperSQL}$	Relational Database Engine	SourceForge	4017	2009	23.50	12.99
	D_{Jetty}	Web Container and Clients	Github	7661	3831	23.48	12.68
	D_{JUnit}	Unit Testing Framework	Github	1128	564	17.44	7.94
	D_{Log4j}	Logging Utility	Apache	3052	1526	20.16	10.75
	D_{Maven}	Build Automation Tool	Github	1901	951	21.75	11.70
	D_{Quartz}	Job Scheduler	Github	1065	533	22.87	11.22
	D_{Roller}	Blog Server	Apache	2666	1333	25.24	12.09
	D_{Spring}	J2EE Framework	Github	14345	7173	20.27	10.61
	D_{Struts}	Web Application Framework	Github	3742	1871	21.45	11.78

^a D_{Dorn} contains code snippets written in Java, Python, and CUDA languages. We concern only the Java samples.^bAll lines in the text of source code are counted, including blank and comment lines.

from various sources (e.g., Github¹⁰ and SourceForge¹¹) across multiple application domains (e.g., web server and testing tool), this is to increase the generalizability of our findings. Similar to the approach used in the prior study [99], we extract from the selected projects all complete code entities (i.e., methods) whose size is between 10 and 50 lines. Afterwards, we calculate the MD5 hash for each file to remove duplicates. As shown in the second part of Table 4.1, we obtain a total of 51831 code snippets. The next step is to classify them as readable or unreadable. According to the definition of code readability, the most accurate method to judge whether a piece of source code is readable is to conduct a large-scale survey involving as many domain experts as possible. The process is inevitably labor-intensive and time-consuming. Considering that we have more than 50,000 code snippets to evaluate, the human assessment approach is practically impossible. We thus propose an alternative method to help classify these samples.

Lee et al. [66] found that source codes that significantly violate programming guidelines/conventions/styles [48, 75] are less readable than those that do not. Buse et al. [17] showed that code readability correlates strongly with external notions of software quality (e.g., defect density and code churn). According to their conclusions, we rely on the total rule violations to obtain a rough estimate of whether a source code is readable. Specifically, we count the number of rule violations for each file making use of automated tools PMD¹² and CheckStyle¹³. PMD examines for common programming flaws like unused variables and empty catch blocks, whereas CheckStyle detects whether a Java code complies with a programming standard. We rely on PMD and CheckStyle to help annotate the newly collected code snippets. The reasons for using third-party tools rather than previous models are as follows: 1) Previous studies [17, 89, 42, 99] have not released their source code for

¹⁰<https://github.com>

¹¹<https://sourceforge.net>

¹²<https://pmd.github.io>

¹³<http://checkstyle.sourceforge.net>

public use, so there is no efficient way to annotate code snippets based on their models. 2) PMD and CheckStyle can cover many features that have been found to be associated with code readability, for instance, comment lines, naming conventions, and defect density. 3) We consider that the use of external tools is likely to be fair and objective, because previous models usually lean towards a certain kind of features. For instance, Buse et al.’s Model [17] involved only layout features (e.g., the number of blank lines), whereas Scalabrino et al.’s Model [99] attached greater importance to textual features (e.g., the readability of comments).

For each project, we consider the top 25% of code snippets with more rule violations as the *Unreadable* group and the bottom 25% of code snippets with fewer rule violations as the *Readable* group. The average violations for the two comparison groups are 84.84 and 30.17, respectively. In this way, we obtain an additional 25,000 samples to adequately train the proposed ConvNets. A statistical summary of the datasets is presented in Table 4.1. Besides, the pie chart in Figure 4.6 shows the distribution of our data sources. Because the new dataset (D_{OSS}) has not been validated by human annotators, it will be used only for tentative explorations (see Section 4.5 for details).

Note that this method is only applicable for dataset construction (i.e., for identification of two groups of *relatively* readable/unreadable code snippets). It is unreasonable to use it for code readability classification. In other words, when given a new instance, we cannot determine its readability degree based merely on its number of rule violations.¹⁴

¹⁴Code readability is essentially a subjective concept [17] that has been found to be associated with various factors such as comment lines [1], naming conventions [12, 101], defect density [42], and rule violations [66]. However, statistical associations do not imply causation. We cannot equate code readability with any of these factors [115, 108].

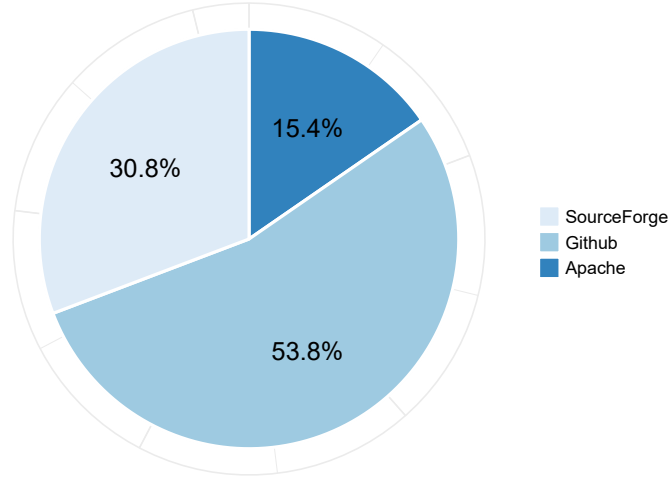


Figure 4.6: The Distribution of Data Sources

4.3.3 Evaluation Metrics

To measure classification results, we use accuracy and f-measure as the evaluation metrics, which have commonly been used in previous code readability studies [17, 42, 99]. The higher the metric value, the better the model performance.

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + False\ Positive + True\ Negative + False\ Negative} \quad (4.5)$$

$$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall} \quad (4.6)$$

where *True Positive* is the number of readable instances that are correctly classified as readable, *True Negative* is the number of unreadable instances that are correctly classified as unreadable, *False Positive* is the number of unreadable instances that are wrongly

classified as readable, *False Negative* is the number of readable instances that are wrongly classified as unreadable, *Precision* refers to $\frac{True\ Positive}{True\ Positive + False\ Positive}$, and *Recall* refers to $\frac{True\ Positive}{True\ Positive + False\ Negative}$. In summary, accuracy shows the ratio of correctly classified instances, whereas f-measure shows the weighted harmonic mean of *Precision* and *Recall*.

4.4 Results

In this section, we present the experimental results according to each research question. For simplicity, we denote the previous datasets (see details in the first part of Table 4.1) as D_{CRS} (CRS stands for Code Readability Survey) and the additional datasets (see details in the second part of Table 4.1) as D_{OSS} (OSS stands for Open Source Software Projects). Note that in RQ1, RQ2, and RQ3, our evaluations use only D_{CRS} for a fair comparison, whereas both D_{CRS} and D_{OSS} are used in RQ4. To help mitigate the problem of over-fitting, we perform 10-fold cross-validation in all RQs. The process is composed of five steps: 1) randomly split the dataset into 10 approximately equal folds; 2) use one fold for testing and the remaining folds for training; 3) classify each code snippet in the test set using the model built on the training set; 4) repeat this process with each fold as the test fold in turn; 5) calculate the average accuracy of the model.

4.4.1 Research Question 1

RQ: Do different parameter settings affect the model performance?

In this RQ, we detail the tuning process with respect to each hyperparameter.

- **Filter Size**

The filter size corresponds to the identifiers $k_1 \times k_2$ (i.e., *width* $\times$ *height*) in Figure 4.4. To investigate the effect of the filter size, we first fix the number of feature maps as a

constant (here, 100). Considering that code readability is context sensitive (just like natural language), we set the width k_1 as the maximum line length, which varies by each ConvNet. Following the approach given by Zhang et al. [117], we then fine-tune k_2 .

We begin by identifying the best single filter size. Specifically, we first select a finite set of values for k_2 and then evaluate each ConvNet’s performance regarding mean accuracy. Note that we bias our selection to small values to capture low-level features and because the average LOC in D_{CRS} is not very large (see Table 4.1 for details). The results are presented in Table 4.2. The best single filter sizes for $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$ are 2, 2, and 9, respectively.

Table 4.2: Effect of the Single Filter Size

Filter Size	ConvNet _{CR}	ConvNet _{TR}	ConvNet _{NR}
1	85.5%	77.3%	67.3%
2	85.8%	84.0%	58.8%
3	79.8%	76.0%	58.5%
4	83.0%	82.3%	64.0%
5	78.3%	80.5%	63.8%
6	75.8%	79.0%	66.5%
7	74.5%	78.0%	69.8%
8	76.0%	78.3%	67.3%
9	76.8%	80.5%	74.3%
10	76.5%	83.5%	72.5%

Next, we explore the effect of multiple filter sizes. As advised by Zhang et al. [117], we choose a set of values near the best single filter size and then perform the same evaluations as above. Table 4.3 demonstrates our selections and the corresponding results (the best results are highlighted in bold). From Table 4.3, we decide to use (2,2,2), (2,2,2), and (8,9,10) as the filter sizes for $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$, respectively.

- **Number of Feature Maps**

Table 4.3: Effect of the Multiple Filter Size^a

ConvNet _{CR}		ConvNet _{TR}		ConvNet _{NR}	
Filter Size	Accuracy	Filter Size	Accuracy	Filter Size	Accuracy
—	—	—	—	(7, 8, 9)	69.5%
(1, 2, 3)	86.5%	(1, 2, 3)	75.5%	(8, 9, 10)	75.5%
(2, 3, 4)	81.5%	(2, 3, 4)	80.0%	(9, 10, 11)	74.5%
(2, 2, 2)	88.0%	(2, 2, 2)	81.0%	(9, 9, 9)	69.5%

^aTo have ConvNets with identical architectures, we consider only the set of size three.

Based on the selected filter size, we now fine-tune the number of feature maps. According to Zhang et al. [117], the value to be evaluated should not exceed 600. A much longer time is required to train the model if the number of feature maps grows beyond 600, but the model performance improves only a little, so the process is not cost-effective. In addition, the model performance may even worsen due to over-fitting. We thus comply with their suggestions and consider the following values: $\{10, 50, 100, 200, 400, 600\}$. The mean accuracy is plotted in Figure 4.7 regarding each ConvNet. It can be observed that 100 is the optimal number of feature maps for $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$.

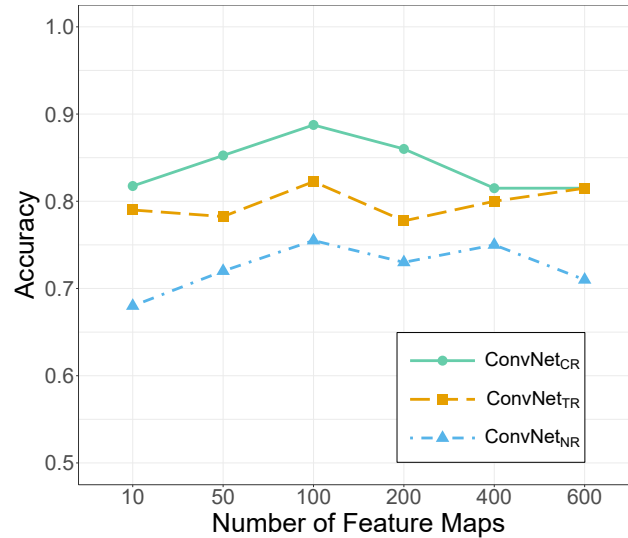


Figure 4.7: Effect of the Number of Feature Maps

Taken together with Table 4.2, Table 4.3, and Figure 4.7, we notice that the model performance varies on a small scale (less than 10 percentage points), which indicates that our ConvNets are somewhat robust to different parameter settings.

4.4.2 Research Question 2

RQ: What is the benefit of the ensemble architecture?

We propose an ensemble architecture that integrates $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$. Because each ConvNet is constructed with a different view (see details in Figure 4.2 and Figure 4.5), we expect the resulting model to be more generic and thus help improve the classification performance.

We present the evaluation results in the first row of Table 4.4. It can be observed that DeepCRM provides the best performance, with 83.8% accuracy and 83.5% f-measure, which validates the effectiveness of our ensemble method. The last two rows of Table 4.4 are used to display the results of an exploratory experiment on D_{OSS} . Further details are provided in Section 4.5.

Table 4.4: Evaluation of the Ensemble Architecture

Dataset	ConvNet _{CR}		ConvNet _{TR}	
	Accuracy	F-Measure	Accuracy	F-Measure
D_{CRS}	82.3%	81.0%	77.3%	75.3%
$D_{CRS} + D_{OSS}$	88.9%	89.4%	74.3%	77.8%
Improvement	8.0%	10.4%	-3.9%	3.3%

Dataset	ConvNet _{NR}		DeepCRM	
	Accuracy	F-Measure	Accuracy	F-Measure
D_{CRS}	71.8%	70.9%	83.8%	83.5%
$D_{CRS} + D_{OSS}$	51.5%	54.8%	91.5%	91.7%
Improvement	-28.3%	-22.7%	9.2%	9.8%

Note that the evaluations are conducted on a hold-out test set from D_{CRS} , which has not been used for hyperparameter optimization (RQ1). Therefore, the accuracy obtained here is generally not equal to that in RQ1.

4.4.3 Research Question 3

RQ: How accurate is DeepCRM in code readability classification when compared to traditional models?

The aim of this RQ is to determine the extent to which DeepCRM can predict human readability judgments compared to previously reported models (i.e., M_{Buse} , $M_{Posnett}$, M_{Dorn} , $M_{Scalabrino}$, and M_{All}). As shown in Figure 4.8, DeepCRM outperforms all competitors (the improvement in accuracy ranges from 2.4% with respect to M_{All} to 17.2% with respect to $M_{Posnett}$).

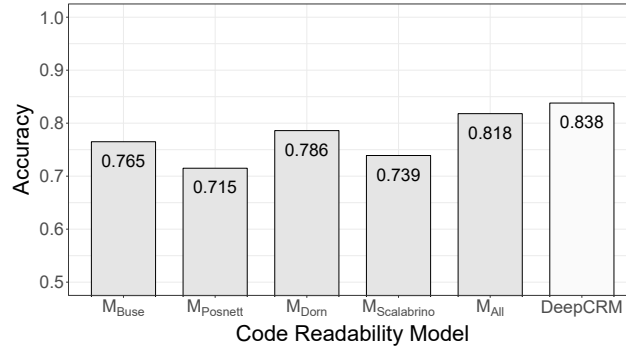


Figure 4.8: Comparison between DeepCRM and the Traditional Code Readability Models

In fact, the goal of this study is not to obtain a significant performance gain/improvement, but rather to explore the potentiality of deep learning techniques in the important task of code readability classification in a more automated fashion. Specifically, our approach eliminates the need for manual feature engineering and can learn various levels of features automatically from the source code. As a result, our approach

provides a slightly improved performance against traditional models (or at the very least, as good as traditional models), which confirms the promising future of applying deep learning techniques to the field of code readability for automatic classification.

4.4.4 Research Question 4

RQ: What is the time and space cost of training the proposed ConvNets?

Training Deep Neural Networks usually consumes a considerable amount of resources. However, most deep learning studies do not report their efforts [47]. To provide a reference for future studies, we explicitly recorded our time and memory space cost during the training process. The results are given in Table 4.5.

Table 4.5: Average Time and Space Cost per Epoch^a

Dataset	ConvNet _{CR}		ConvNet _{TR}	
	Time (Second)	Memory (MB)	Time (Second)	Memory (MB)
D_{CRS}	0.22	11.74	0.20	11.10
$D_{CRS} + D_{OSS}$	32.40	15.50	32.41	13.75
Ratio	144.85	1.32	163.20	1.24

Dataset	ConvNet _{NR}		DeepCRM	
	Time (Second)	Memory (MB)	Time (Second)	Memory (MB)
D_{CRS}	0.23	11.52	0.47	12.14
$D_{CRS} + D_{OSS}$	29.74	14.70	71.46	19.05
Ratio	129.97	1.28	152.14	1.57

^a An epoch is an iteration through the entire training set.

All of our experiments are conducted on a server with Intel Xeon CPU E5-4620 2.20 GHz (32 Cores), 128 GB RAM. As shown in Table 4.5, the memory space cost varies within a small range (from 11.10 MB to 19.05 MB), mainly because the number of training

instances used in one iteration (i.e., the batch size) is fixed. As for the time cost, training on D_{CRS} requires approximately 0.22 to 0.47 seconds for each epoch, whereas the value increases to 29.74 to 71.46 seconds if we further introduce D_{OSS} , the ratio reaches 129.97 to 163.20.

Admittedly, our approach has the downside of being time-consuming during the training process. However, once training is complete, our approach can be used directly at an acceptably low time cost (the prediction time is less than 0.001 seconds for a new instance), which confirms its practical applicability.

4.5 Exploratory Experiments

This section presents several exploratory experiments. Specifically, we first examine the benefits of adopting a large training set (D_{OSS}). Then we investigate whether the addition of annotator information can further improve the accuracy of code readability classification. Finally, we experiment with a different model architecture.

4.5.1 Exploratory Experiment 1

RQ: How much improvement can be expected from a large training set?

Convolutional Neural Networks (ConvNets) risk over-fitting if the training set is not sufficiently large. To adequately train the proposed ConvNets, we collect more than 25,000 code snippets from open source software projects (see details in the second part of Table 4.1). In this exploratory experiment, we evaluate whether training on the new dataset (D_{OSS}) can help improve the model performance.

We train $ConvNet_{CR}$, $ConvNet_{TR}$, $ConvNet_{NR}$, and DeepCRM on D_{OSS} , and compare their performances with those trained using D_{CRS} (the old dataset provided by previous

code readability studies [17, 42, 99]). The evaluation results are presented in the last two rows of Table 4.4. As shown, $ConvNet_{CR}$ and DeepCRM have better performance on D_{OSS} than on D_{CRS} . The improvement in accuracy is 8.0% and 9.2%, respectively. $ConvNet_{TR}$ yields comparable results, whereas the performance of $ConvNet_{NR}$ becomes much worse, probably because $ConvNet_{NR}$ is trapped in a local optimum.¹⁵ We will further investigate this problem in a future study.

In this exploratory experiment, we examine the benefits of adopting a large training set (D_{OSS}). Note that because D_{OSS} has not been validated by human annotators, the current results should be considered preliminary.

4.5.2 Exploratory Experiment 2

RQ: Does annotator information contribute to the model performance?

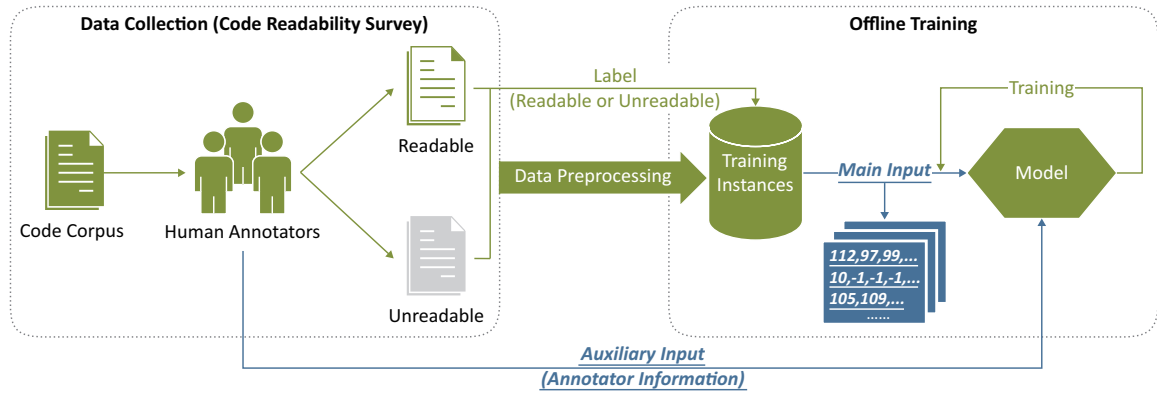


Figure 4.9: An Auxiliary Input for Model Training

As shown in Figure 4.9, the result of a code readability survey is usually a set of code snippets accompanied by readability scores, which serves as the main input for model training. Considering that the obtained mean-opinion-scores may not be equally reliable, we

¹⁵Another possibility is that node-level representation is still too abstract to preserve sufficient information for code readability classification.

propose to apply the information of human annotators as the auxiliary input to our model. Although a variety of metrics can be considered, in this study we especially focus on the following two:¹⁶ 1) the number of annotators who have rated a code snippet (code snippets that have been rated by more annotators are more reliable than those with less); 2) the standard deviation of readability scores on a code snippet (a lower standard deviation indicates greater consistency among annotators).

Specifically, we examine whether there exists significant difference between the performance of our model with and without annotator information using robust test statistics recommended by Kitchenham et al. [59], i.e., the Brunner-Munzel test (significance level $\alpha = 0.05$) [16, 81] and Cliff’s Delta [24].

We observe that without annotator information, the proposed model is able to correctly classify 80.19% of code snippets in our corpus. When annotator information is included, the accuracy increases to 82.76%. The *p-value* for the Brunner-Munzel test is 0.016 (<0.05) and the *d-value* for Cliff’s Delta is 0.641 (>0.474), implying a significant difference with a large effect size.

This experiment is the first attempt to explore the impact of annotator information in the context of code readability classification. However, we involve only two metrics. In future work, we plan to examine additional factors, for instance, the annotators’ development experience (the opinions of domain experts are more reliable than those of university students).

4.5.3 Exploratory Experiment 3

RQ: Can the Inception architecture be used to improve the model performance?

¹⁶The main reason is that the information of human annotators has not been properly recorded by prior work [17, 42, 99]. As a result, we have only limited data to work with.

Inspired by GoogLeNet [107],¹⁷ we adapt the Inception architecture for code readability classification. The primary reasons are as follows: 1) The Inception architecture is proven to be effective in capturing multi-scale information from the input data; 2) When building a ConvNet layer, there is a challenge of choosing either a convolution or a pooling operation. The Inception architecture enables the simultaneous use of both.

Figure 4.10 illustrates the Inception module modified for the task of code readability classification, which contains a set of convolutions and poolings alongside each other. Specifically, we make two modifications based on Inception-v6 [106]. First, we apply the same receptive field in each module for computational savings. We bias our selection to minimal values (here, 2) to capture lower-level features. Second, because code readability is context sensitive (just like natural language), the width of the filter size on top levels is deliberately set as the maximum line length (MLL).

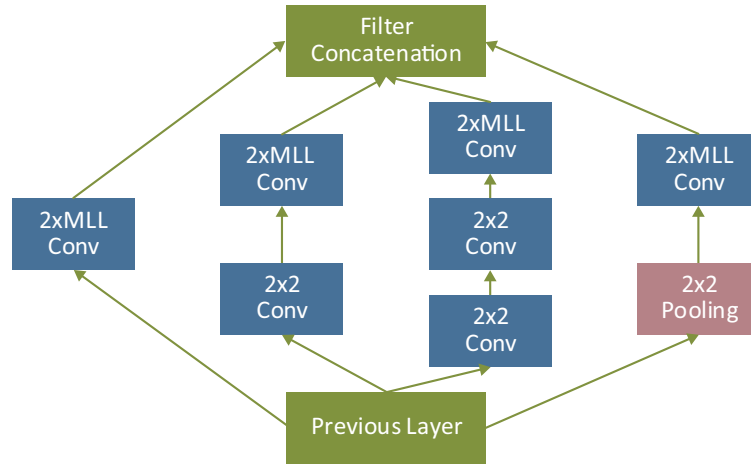


Figure 4.10: A Modified Inception Module

In particular, the feature learning network consists of Inception modules, whereas the classification network consists of fully-connected layers followed by a Softmax layer. To introduce nonlinearity and accelerate the training process, ReLU (Rectified Linear Unit)

¹⁷GoogLeNet is a deep ConvNet constructed by stacking Inception modules.

activation and batch normalization are applied after each convolutional layer. In addition, we involve the dropout strategy to reduce over-fitting, which randomly disables neurons during the learning phase.

To evaluate the efficacy of the Inception architecture, we compare our model with five state-of-the-art code readability models (i.e., M_{Buse} , $M_{Posnett}$, M_{Dorn} , $M_{Scalabrino}$, and M_{All}). We apply different machine learning techniques as the underlying classifier for each model, namely Logistic Regression (LR), Bayesian Network (BN), Multilayer Perception (MLP), Sequential Minimal Optimization (SMO), and Random Forest (RF).

The accuracy achieved by each model on three publicly available datasets is provided in Table 4.6. The results show that the Inception architecture-based model yields higher accuracy against its competitors across all datasets.

4.6 Discussion

In this section, we discuss the advantages and disadvantages of applying deep learning techniques to the field of code readability classification. Then we describe the potential threats to the validity of our results.

4.6.1 Advantages and Disadvantages of Deep Learning

The experimental results show that the proposed ConvNets perform better than the traditional models [17, 42, 89, 99] for code readability classification (the improvement in accuracy ranges from 2.4% to 17.2%). In this section, we discuss why our approach achieves the best performance.

- First, we eliminated the need for manual feature engineering. Inspired by image recognition, we treat a source code as a matrix of symbols and leverage ConvNets

Table 4.6: Accuracy Achieved by Code Readability Models

Model		D_{Buse}	D_{Dorn}	$D_{Scalabrino}$
M_{Buse}	LR	0.810	0.786	<u>0.705</u>
	BN	0.760	0.750	0.625
	MLP	0.760	0.742	0.665
	SMO	<u>0.820</u>	<u>0.797</u>	0.670
	RF	0.800	0.781	0.680
$M_{Posnett}$	LR	<u>0.780</u>	<u>0.728</u>	0.660
	BN	0.760	0.681	<u>0.695</u>
	MLP	0.770	0.703	0.655
	SMO	0.770	0.711	0.680
	RF	0.770	0.703	0.605
M_{Dorn}	LR	<u>0.800</u>	<u>0.800</u>	<u>0.755</u>
	BN	0.670	0.747	0.640
	MLP	0.720	0.725	0.685
	SMO	0.770	0.767	0.725
	RF	0.750	0.744	0.690
$M_{Scalabrino}$	LR	0.740	<u>0.772</u>	<u>0.680</u>
	BN	0.530	0.681	0.640
	MLP	<u>0.770</u>	0.742	0.665
	SMO	0.720	0.717	0.650
	RF	0.720	0.742	0.655
M_{All}	LR	0.790	<u>0.839</u>	<u>0.795</u>
	BN	0.720	0.761	0.710
	MLP	0.730	0.778	0.700
	SMO	<u>0.810</u>	0.806	0.770
	RF	0.770	0.789	0.690
An Inception Architecture-Based Model		0.825	0.842	0.822

to automatically learn features directly from the input data. Our approach has the advantage of requiring no human intervention and thus can effectively avoid personal biases and neglects of certain features.

- Second, we proposed a representation strategy (with different granularities) to preserve the source code's original information. Despite little tuning of the hyperparameters, a simple ConvNet can achieve fairly good results based on the resulting integer matrices (RQ1 and RQ2).
- Third, we integrated multiple ConvNets with an ensemble architecture to improve model generality and applicability. The experimental results show that the ensemble model can surpass the state-of-the-art, achieving 83.8% accuracy and 83.5% f-measure (RQ2 and RQ3).
- Finally, a new dataset with more than 25,000 code snippets was used to fully train the proposed ConvNets. We observe further improvements by about 9.2% in accuracy and 9.8% in f-measure (The exploratory experiment presented in Section 4.5).

By eliminating the need for manual feature engineering (one of the most time-consuming parts of traditional machine learning), our approach provides improved performance, confirming the feasibility of deep learning techniques for code readability classification. Although deep learning is an important area of research with many promising applications, several drawbacks and limitations accompany its use.

- **Data Dependency:** Deep learning requires a large corpus of data with which to work, which may not always be available. Without large, well-maintained datasets, deep learning may only be able to yield results comparable to those of traditional machine learning (RQ3).
- **Time Cost:** Because deep learning inherently involves a large amount of matrix multiplication operations (e.g., Equation 4.1 and Equation 4.4), the training of Deep Neural Networks is usually computationally expensive (RQ4). This leads to the next disadvantage.

- **Hardware Dependency:** Unlike traditional machine learning, deep learning is quite resource-demanding and depends heavily on high-performance machines (preferably with powerful GPUs).
- **Model Interpretability:** Deep learning is in principle a black-box approach. Unlike traditional machine learning (e.g., decision trees), deep learning is incapable of explaining why a certain decision has been reached. In other words, we cannot map decisions back to individual features.

Although deep learning has achieved impressive (and often state-of-the-art) results in multiple domains, it is certainly not flawless. The caveat is that deep learning should not be applied to all possible applications without weighing its pros and cons.

4.6.2 Threats to Validity

In this section, we describe a number of issues that affect the validity of our work.

4.6.2.1 Internal Validity

- **Data Quality**

The most obvious threat is the quality of the training data. If the data is prone to errors, then it will be reflected in our results. Broadly, we have two data sources. A minor part of our dataset (approximately 0.4%) comes from previous code readability studies [17, 42, 99], and the rest is taken from open source software projects. We outline the possible threats subject to each data source.

To abstract people's perceptions of whether a source code is readable, the most frequently used method is to conduct a survey. During the survey process, participants are

required to rate a number of code snippets based on their knowledge and experience. Considering that the process in itself is neither amusing nor rewarding, it is possible that the participants will feel little motivation to contribute, resulting in negative behaviors such as random responses [43], which can significantly affect the quality of the survey data. In addition, most participants involved in code readability surveys are students with less programming experience than industrial practitioners. Although the use of college students imposes some limits on data validity, it is not believed that the overall results are skewed. Moreover, prior work has validated their datasets with inter-rater reliability tests (i.e., Cronbach’s alpha [27]). Considering the positive results (e.g., 0.96 for D_{Buse} and 0.98 for $D_{Scalabrino}$), we regard these datasets as the reliable oracle.

Given that the literature contains only a few human-annotated code snippets, which may not be sufficient to train the proposed ConvNets, we turn to open source Java projects. Although we sampled projects of various sizes and from various communities, they cannot represent all software projects. Additionally, we biased our selection to popular projects to ensure the size of the dataset. It is possible that other open source (or industrial) projects could yield different conclusions. Besides, we used automated tools to roughly classify the code snippets. Although our approach is defensible, further study is needed to justify the new dataset.

• Sample Size

Another limitation is that when addressing RQ1, RQ2, and RQ3, we use only D_{CRS} rather than $D_{CRS} + D_{OSS}$ for two primary reasons. First, to achieve a fair comparison, our ConvNets must be trained on the same dataset as in the previous studies. Second, the quality of D_{CRS} , but not D_{OSS} , has been validated by human annotators.¹⁸ However, the sample size of D_{CRS} is admittedly small, which may threaten the internal validity of our results.

¹⁸Accordingly, D_{OSS} has only been used in RQ4 for a tentative exploration.

Given that the proposed ConvNets can provide consistently high and stable performance, we consider the use of D_{CRS} in RQ1, RQ2, and RQ3 to be acceptable and feasible.

4.6.2.2 Construct Validity

In our experiments, we mainly use accuracy and f-measure to evaluate the effectiveness of the proposed model. However, other performance measures such as precision, recall, and AUC (Area Under the ROC Curve) can also be considered. We leave this for future studies.

Chapter 5

Conclusions and Future Work

This thesis has been devoted to the study of code readability by using several latest techniques. In this chapter, we summarize our main contributions and discuss some promising future directions.

5.1 Summary

In this thesis, we first presented the design, implementation, and evaluation of GamiCRS in Chapter 3, which is a novel system that incorporates gamification techniques and infrastructure into the context of code readability survey. Specifically, we made the following main contributions:

- A reusable gamification system is proposed to enhance code readability survey (GamiCRS), which is well-founded in Self-Determination Theory and User-Centered Design. To the best of our knowledge, this is the first work giving careful consideration to participants' needs and goals when engaging in a survey process.

- A large-scale field experiment is performed to validate the practical feasibility of GamiCRS and help to delineate future working directions. A questionnaire approach is utilized to investigate participants' impressions and attitudes toward the acceptance of GamiCRS. The empirical analysis shows that GamiCRS is effective for improving user experience and data quality.

GamiCRS can help researchers overcome the complexities of gamification design and implementation. Also, it provides a relaxing and rewarding survey experience for participants. We hope that our study can interest and encourage further research regarding gamification, which we believe is a promising methodology for changing behavioral patterns and for supporting a wide field of possible applications in software engineering research.

After that, we introduced a completely new method for code readability classification based on Convolutional Neural Networks (ConvNets) in Chapter 4. We first proposed a simple and customizable representation strategy (with different granularities) to transform source codes into integer matrices as the input to ConvNets. We then implemented DeepCRM, a novel code readability model consisting of three separate ConvNets with identical architectures. Our approach is able to learn various levels of features automatically from the source code. To evaluate DeepCRM, we conducted a series of experiments. The results show that DeepCRM can outperform state-of-the-art models (i.e., the models of *Buse et al.* [17], *Posnett et al.* [89], *Dorn et al.* [42], and *Scalabrino et al.* [99] and *A Comprehensive Model* [99]). The improvement in accuracy ranges from 2.4% to 17.2%.

Our work serves as the first step toward deep learning-based code readability classification. We hope that the promising results will interest and encourage more in-depth research in this new field.

To enable critical or extended analyses, our dataset and source code are publicly available.¹ Specifically, we have published two folders. The folder *Code Snippet Repository* contains only D_{OSS} (the new datasets retrieved from open source Java projects), while D_{CRS} (the old datasets provided by previous code readability studies) has been available online for public use (please refer to [17, 42, 99]). The folder *Source Code* contains Python programs we have developed, which can be used to: 1) count the number of rule violations for code snippets; 2) transform code snippets into integer matrices according to the specified granularity (i.e., Character-Level, Token-Level, or Node-Level Representation).

5.2 Future Work

There are several directions for future research. The top priority is to further improve the model performance.² To achieve this, we can fine-tune model hyperparameters (e.g., filter size and dropout rate), experiment with different configurations (e.g., different optimizers and loss functions), or attempt a different model architecture. Given that the data quality can significantly affect the model quality, we also plan to construct a better training set with more reliable data (i.e., human-annotated code snippets), which can hopefully improve our results. Another option is to optimize the strategy used to represent the source code, which is one of the major challenges in deep learning-based program analyses.

In addition, existing studies (including our own) tend to classify a piece of source code as either readable or unreadable. However, readability is not necessarily a binary decision. In the future, we plan to investigate the possibility of measuring code readability in terms of continuous values to make the output more precise and interpretable (just like *Reading Grade Level* for natural languages [45]).

¹<https://github.com/CityU-QingMi/DeepCRM>

²The model performance here refers to not only accuracy but also robustness, reliability, generality, etc.

In this study, we treat a code snippet (a separate file) as an atomic unit for code readability classification, which may be somewhat coarse-grained. We expect that in future our approach can precisely locate unreadable regions rather than forcing users to search through the entire file. This drawback actually applies to all existing code readability studies.

Currently, our approach allows only Java code. We intend to extend it to other programming languages. We consider the plan to be feasible because the only language-dependent part is the lexical analysis process presented in Figure 4.2. All remaining steps are adaptable with slight adjustments.

Additionally, there is a need to find out what exact factors make a code more or less readable. We plan to further explore this open question with the help of eye tracking technologies. Once completed, we are able to provide targeted suggestions to developers on how to improve their code.

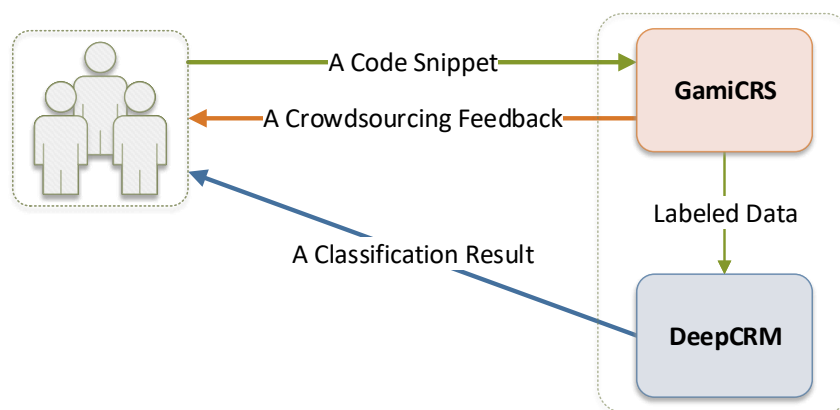


Figure 5.1: A Novel Framework for Code Readability Classification

Another direction for future work is to apply the proposed model in a real-world situation. For instance, we can integrate DeepCRM into modern IDEs (Integrated Development Environments) or version control systems to enable developers to continually monitor the

readability of their code. Moreover, we intend to combine DeepCRM with GamiCRS. As shown in Figure 5.1, after uploading a code snippet, users of GamiCRS can receive: 1) a crowdsourcing feedback from GamiCRS; 2) a classification result from DeepCRM. On the other hand, DeepCRM, in turn, can be fine-tuned using data collected from GamiCRS.

Appendix A

Acronyms And Abbreviations

(Listed in alphabetical order)

AST	Abstract Syntax Tree
ATGDE	Attitude Toward Game Design Elements
AUC	Area Under the ROC (Receiver Operating Characteristic) Curve
BN	Bayesian Network
ConvNet	Convolutional Neural Network
CR	Character-Level Representation
CRS	Code Readability Survey
GDE	Game Design Element
IDE	Integrated Development Environment
LR	Logistic Regression
MLL	Maximum Line Length
MLP	Multilayer Perception

NASA-TLX	NASA Task Load Index
NR	Node-Level Representation
OSS	Open Source Software Project
PBL	Points, Badges, and Leaderboards
PEOU	Perceived Ease of Use
PFM	Programming Format Metrics
PLM	Programming Language Metrics
PRM	Programming Readability Metrics
PU	Perceived Usefulness
ReLU	Rectified Linear Unit
RF	Random Forest
SDT	Self-Determination Theory
SE	Software Engineering
SLOC	Source Lines of Code
SMO	Sequential Minimal Optimization
TAM	Technology Acceptance Model
TR	Token-Level Representation
US	User Satisfaction

Appendix B

Using Eye Tracking Technology to Analyze the Impact of Stylistic Inconsistency on Code Readability

As no conclusive evidence in literature has shown that how and to what extent stylistic inconsistency degrades code readability, we carefully plan a controlled experiment to bridge this research gap, in which eye tracking technology is introduced to capture participants' cognitive responses and mental processes. We believe that our results will be beneficial for both researchers and practitioners to better understand the stylistic inconsistency issue, and take further actions to improve code readability as the ultimate goal.

B.1 Context and Problem Statement

Programming style can be viewed as a kind of personal preferences when writing the source code, which commonly relates to the program's visual appearance (e.g., naming of variables, how and when to use comments, indentation, and alignment). Due to different experiences and personalities, developers usually have their very own programming styles, yet these individually preferred programming styles may well conflict with each other. When collaborating, it is highly likely to degrade the readability and maintainability of the source code being produced. The problem is referred to as *stylistic inconsistency* in our previous research [75].

A number of research efforts have focused in the area of programming style. However, to the best of our knowledge, there is little sound and solid evidence of how and to what extent can stylistic inconsistency impact the readability and maintainability of the source code, which is the fundamental problem that should be resolved first in this context. Thus, we carefully plan an empirical experiment with which to bridge the research gap. The expected contribution of our work is two-fold:

- Provide solid evidence of the impact of stylistic inconsistency on source code comprehension.
- Present a useful and reusable research framework for analyzing the influences different stylistic factors have upon code readability.

To accurately reflect developers' cognitive efforts required to read and understand a given program, we introduce eye tracking technology to capture ocular indices (e.g., gaze position and pupil size) as objective measurements. Basically, eye movements can be characterized as a sequence of fixations and saccades.

- **Fixation:** a spatially stable gaze on an object of interest that lasts for a certain period of time. The information acquisition and processing mainly occurs during fixations, which is the most relevant index to our experiment.
- **Saccade:** a quick and simultaneous movement of both eyes from one location to another.

Eye tracking technology is an affordable and easy-to-use approach enabling us to identify what a developer is indeed viewing, for how long, and in what order, which already has broad applications in the domain of usability evaluation, cognitive interface design, and so forth. However, there are only a few studies introduce eye tracking techniques and tools into the context of software engineering research. Rodeghero et al. [91] presented a novel tool for source code summarization based on the findings of eye tracking study. Fritz et al. [46, 79] investigated how biometric sensors (i.e., eye tracking, EDA, and EEG) can be used to measure developers' perceived difficulty while working on a change task. Considering their success, we employ eye tracking technology in this study. The fine-grained measurements can establish effective relations between cognitive processes and program comprehension, which provide more credible evidence for our analysis as compared to the tradition techniques such as survey and think-aloud strategy.

B.2 Proposed Approach

In this section, we present a detailed description of the controlled experiment, which is designed to empirically investigate the impact of stylistic inconsistency on code readability.

```

1  class Task {
2  public:
3      Task(int* src, int len) : src_(src), len_(len) {
4      }
5      int process_data(int key) {
6          int left = 0, right = len_ - 1, middle;
7
8          while (left <= right) {
9              middle = (left + right) / 2;
10             if (key == src_[middle]) {
11                 return middle;
12             }
13             else if (key < src_[middle]) {
14                 right = middle - 1;
15             }
16             else if (key > src_[middle]) {
17                 left = middle + 1;
18             }
19         }
20         return -1;
21     }
22 private:
23     int* src_;
24     int len_;
25 };

```

(a) Google Style Guides

```

1  class CTask
2  {
3  public:
4      CTask(int* pSrc, int nLen) : m_pSrc(pSrc), m_nLen(nLen)
5      {
6      }
7      int ProcessData(int nKey)
8      {
9          int nLeft = 0, nRight = m_nLen - 1, nMiddle;
10
11         while (nLeft <= nRight)
12         {
13             nMiddle = (nLeft + nRight) / 2;
14             if (nKey == m_pSrc[nMiddle])
15             {
16                 return nMiddle;
17             }
18             else if (nKey < m_pSrc[nMiddle])
19             {
20                 nRight = nMiddle - 1;
21             }
22             else if (nKey > m_pSrc[nMiddle])
23             {
24                 nLeft = nMiddle + 1;
25             }
26         }
27         return -1;
28     }
29 private:
30     int* m_pSrc;
31     int m_nLen;
32 };

```

(b) Windows Coding Style Conventions

Figure B.1: Style Variations

B.2.1 Independent Variables

The primary independent variable in our experiment is the presence or absence of stylistic inconsistency. Note that programming style is essentially a multifaceted concept [75]. In this study, we focus solely on typographic characteristics that represent the physical layout of the source code (e.g., indentation, alignment, and spacing).

Figure B.1 serves as a visual demonstration of the primary independent variable, which depicts the variations of typographic style on a given algorithm. The code snippets in Figure B.1a and Figure B.1b conform to Google Style Guides¹ and Windows Coding Style Conventions², respectively. Note that the two programs are lexically similar but quite different in format.

B.2.2 Dependent Variables

As is shown in Table B.1, a group of subjective and objective measurements is proposed to represent code readability, which is the primary dependent variable in this study. In addition to the traditional measurements (e.g., NASA-TLX) that have been widely used in cognitive research, we introduce a novel variable, namely, visual effort. The general consensus is that ocular indices could effectively reflect individuals' cognitive responses and mental processes when solving a given task.

B.2.3 Secondary Variables

Secondary variables are a special type of independent variables that may affect the relationship between the primary independent variables and the dependent variables. In this study, we consider the following secondary variables:

¹<https://github.com/google/styleguide>

²[https://msdn.microsoft.com/en-us/library/windows/desktop/aa378932\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa378932(v=vs.85).aspx)

- **Development Experience:** the level of professionalism. The two possible values are expert and novice according to the number of years involved in programming activity.
- **Style Preference:** the preference of programming style, which is used to determine if familiarity with a particular programming style has an effect on experimental results. The three possible values are Google Style Guides, Windows Coding Style Conventions, and no preference.

B.2.4 Hypothesis

We then generate null hypothesis according to our research objective in a parameterized form [105].

H_0 : There is no significant difference in X between different Y when performing comprehension tasks, where X is one of the dependent variables (i.e., self-report effort, comprehension test, and visual effort), and Y is one of the independent variables (i.e., stylistic inconsistency, development experience, and style preference).

By replacing the parameters, we can obtain the corresponding null hypothesis. For instance, H_0 : There is no significant difference in *Visual Effort* between different *Style Inconsistency* when performing comprehension tasks.

B.2.5 Stimuli and Task Design

The stimuli shown to the participants are a series of code snippets conforming to different programming styles, such as those shown in Figure B.1. As we focus solely on typographic variations, all other respects are deliberately made equal.

Table B.1: Dependent Variables Adopted in this Study and the Corresponding Measurements

Dimension	Dependent Variable	Measurement	Description
Subjective	Self-Report Comprehension Effort	NASA-TLX ^a	Participant ratings for perceived workload.
	Comprehension Test	Accuracy	Correct answer rate for comprehension test.
		Answer Time	Average response time for completing the given task.
	Visual Effort	Fixation	The total number of fixations on the given program adjusted by SLOC ^b .
		Saccade	More fixations indicate less efficiency.
		Gaze Time	The total number of saccades. More saccades indicate more searching effort.
		Pupil Dilation	The amount of time spent viewing the given program adjusted by SLOC ^b . Longer duration indicates more difficulty.
Objective		Blink Rate	The percentage change in pupil diameter. Larger pupil indicates more cognitive effort.
			The relative change (from baseline) in blink rate. Lower blink rate indicates higher mental load.

^aNASA Task Load Index: a widely used rating scale for assessing subjective mental workload.^bSLOC (Source Lines of Code): a count of lines in the text of source code excluding blank and pure comment lines.

To avoid the carryover effects, we carefully plan a between-subjects design. All participants are evenly divided into two groups. In the *Control Group*, participants come to a set of code snippets with a consistent programming style, whereas in the *Treatment Group*, the code snippets alternate with different programming styles to simulate a scenario of stylistic inconsistency. The primary task for participants is to answer the questions accompanied by each code snippet. To alleviate the learning burden, each task is designed to take only a short time (e.g., 1-2 minutes) to complete.

B.2.6 Experimental Procedure

Referring to previous studies (including but not limited to [46, 91]), the procedure of our experiment is designed as follows:

1. Participants are expected to sign an informed consent form.
2. Calibrate eye tracker (i.e., the device used to track eye movement in an unobtrusive way) for each participant due to varying eye characteristics.
3. The experimental procedure and main tasks are introduced to participants, but not the true objective. Also, participants are told that they are free to withdraw from the experiment at any time they want.
4. A pilot experiment is performed to familiarize participants with the equipment and the corresponding workflow. The data obtained in this stage will be discarded without any analysis.
5. In order to guarantee a consistent performance, participants are required to rest 1-2 minutes at intervals for relaxation.

6. For each task, participants are instructed to view a certain code snippet, and then complete a post-hoc comprehension test that consists of 1-2 objective questions. Both the code snippet and the questions are displayed on one screen. Eye movement data is implicitly captured as the participants solving the given task.
7. After each task, participants are required to report their comprehension effort using NASA-TLX.
8. A post-experiment questionnaire is conducted to collect the participants' demographics (e.g., age, gender, ethnicity, education, development experience, and style preference).

During the entire experimental procedure, stage 5-7 may appear several times sequentially.

Appendix C

Stylistic Fingerprinting

C.1 Context and Problem Statement

Programming Style can be viewed as a kind of personal habits when writing the source code, which reflects individual's preferences in terms of physical layout (e.g., alignment and indentation), data structure, algorithm, etc. It tends to become ingrained, which is highly identifiable but difficult to quantify and measure. Although many software engineering methodologies and guidelines are provided, it is common that developers apply their very own programming styles to the source code being produced. These individually preferred programming styles are more comprehensive for themselves, but may well conflict with each other. Thus, the problem of stylistic inconsistency is inevitable during the software development process involving multiple developers, the result is undesirable and that will significantly degrade program readability and maintainability.

A considerable number of previous studies have focused in the area of programming style. For instance, investigating the impact of stylistic variants on program comprehension [11, 76], or coding style-based plagiarism detection [3]. However, there is little focus on

the measurement of stylistic inconsistency. In order to assist practitioners to better understand the inconsistency degree of programming styles within a software project team, we perform an empirical analysis in this study. Specifically, we propose stylistic fingerprints, which are represented as a set of attribute-counting-metrics, in an attempt to characterize different programming styles. Then we adopt the hierarchical agglomerative clustering (HAC) technique to quantitatively measuring the proximity of programming style based on six C/C++ open source projects chosen from different application domains.

The empirical results demonstrate the feasibility and validity of our fingerprinting methodology. Moreover, the proposed clustering procedure utilizing HAC algorithm with dendrograms is capable of effectively illustrating the inconsistency degree of programming style among source files, which is significant for future research.

C.2 A Taxonomy for Stylistic Metrics

Programming style is an intuitive concept, which is difficult to define and measure. Although the stylistic features of source code are less flexible as compared to, for instance, handwriting or literary writing, due to limited keywords and structured format, there is still enough room for developers to establish their own distinctive styles.

Inspired by the work of Krsul et al. [61] and Avgustinov et al. [5], we adopt the concept of fingerprints to characterize different programming styles. A collection of potentially useful metrics is presented in Table C.1 as components to construct stylistic fingerprints as the basis for further analysis.

Considering the large number of stylistic metrics involved in this study, we first propose a classification methodology based on the work from Oman et al. [86, 87] and Krsul et al. [61].

- **Programming Format Metrics (PFM)**

Metrics fall into this category are mainly related to the format (e.g., indentation, alignment and spacing) of the source code. It is necessary for developers to maintain a clear and easy-to-understand physical layout making use of white-space characters (e.g., space, tab, linefeed and carriage return). However, the way the white-space characters being used in the source code is quite different. Note that all metrics in this category could be easily changed by the use of formatting tools (e.g., Pretty Printer¹ and Artistic Style²). Although fragile to some extent, the format is still a distinctive characteristic in terms of measuring stylistic inconsistency.

- **Programming Readability Metrics (PRM)**

This kind of metrics is used to represent the degree of program readability, which has no impact on functionality and efficiency of the source code. In this category, we include such metrics as the ones that measuring naming preference, the proportion of comments and line length. These metrics still closely pertaining to the layout of the source code, but are relatively difficult to change by automatic formatting tools.

- **Programming Language Metrics (PLM)**

Slightly different from the classification as given in Krsul's study [61], this category is designed to involve metrics that are highly correlated with the way developers prefer to implement a particular functionality. For instance, metrics reflecting one's preference of either for, while or do-while loops. It should be noticed that metrics fall into this category may vary considerably when applied to other programming languages.

¹<http://prettyprinter.de>

²<http://astyle.sourceforge.net>

In conclusion, PFM purely correspond to white-space characters that represent horizontal or vertical space in typography. PRM would not be affected by the use of formatting tools on the source code being analyzed. Both of above-mentioned metrics pertain only to physical layout, which is considered highly influential in viewer’s first impression of the program. On the other hand, PLM characterize the way the implementation of certain functionality.

Note that our taxonomy is proposed to provide a clear framework to better assist the further analysis. We do not claim it is flawless. There may be some overlap among the three categories or disagreements about the placement of specific metrics.

C.3 Stylistic Metrics Considered in This Study

A wide variety of previous studies (including but not limited to [8, 39, 61, 87, 90]) influence our choice of stylistic metrics, among which the metrics proposed by Krsul et al. [61] for C programs and Ding et al. [39] for Java programs are the primary ones. Nonetheless, metrics used by those work may be ambiguous, contradictory, even outdated. By reference to those studies, we refine and present a total of 60 metrics in Table C.1 as components to construct stylistic fingerprints for characterizing different programming styles.

To extract these metrics, we develop a Python program (available on-line³) on the basis of `cpplint`, which is an open source lint-like tool released by Google. CSV-formatted datasets are automatically generated after scanning all the source files involved in the software projects.

³<https://github.com/robertyuyang/StylisticFingerprinting>

Table C.1: Stylistic Metrics Adopted in Our Previous Work [75]

Dimension	Metric	Description
Programming Format Metrics (PFM)	PFM01	Percentage of open curly brackets ({}) that are alone in a line.
	PFM02	Percentage of open curly brackets ({}) that are the first character in a line.
	PFM03	Percentage of open curly brackets ({}) that are the last character in a line.
	PFM04	Percentage of open curly brackets ({}) that are in the middle of a line.
	PFM05	Percentage of close curly brackets (}) that are alone in a line.
	PFM06	Percentage of close curly brackets (}) that are the first character in a line.
	PFM07	Percentage of close curly brackets (}) that are the last character in a line.
	PFM08	Percentage of close curly brackets (}) that are in the middle of a line.
	PFM09	Percentage of functions with blank line(s) before close curly brackets (}).
	PFM10	Average white spaces or horizontal tabs to the left side of assignment operators.
	PFM11	Average white spaces or horizontal tabs to the right side of assignment operators.
	PFM12	Average white spaces or horizontal tabs after // or /* symbol.
	PFM13	Percentage of SLOC ^a without indentation.
	PFM14	Percentage of SLOC with white spaces for indentation.
	PFM15	Percentage of SLOC with horizontal tabs for indentation.
Programming Readability Metrics (PRM)	PRM01	Whether the Copyright message appears at the top of the file.
	PRM02	The ratio of the number of blank lines to the number of non-blank lines.
	PRM03	The ratio of the number of comment lines to the number of non-commentary lines (i.e., pure code lines).
	PRM04	The ratio of the number of pure comment lines to the number of comment lines.
	PRM05	The ratio of the number of code lines with inline comments to the number of comment lines.
	PRM06	The ratio of the number of single-line comments (// symbol) to the number of SLOC.
	PRM07	The ratio of the number of multi-line comments (/* and */ pair of symbols) to the number of cal SLOC.

Table C.1 Continued

Dimension	Metric	Description
Programming Readability Metrics (PRM)	PRM08	The ratio of the number of TODO comments to the number of non-blank lines.
	PRM09	Percentage of uppercase characters for function name.
	PRM10	Percentage of lowercase characters for function name.
	PRM11	Percentage of number characters for function name.
	PRM12	Percentage of underscores characters for function name.
	PRM13	Percentage of uppercase characters for variable name.
	PRM14	Percentage of lowercase characters for variable name.
	PRM15	Percentage of number characters for variable name.
	PRM16	Percentage of underscores characters for variable name.
	PRM17	Average function name length.
	PRM18	Average variable name length.
Programming Language Metrics (PLM)	PRM19	Average SLOC per function.
	PRM20	Average line length of SLOC.
	PLM01	Percentage of for in total of for, while and do-while loops.
	PLM02	Percentage of while in total of for, while and do-while loops.
	PLM03	Percentage of do-while in total of for, while and do-while loops.
	PLM04	Percentage of if-else in total of if-else and switch-case branches.
	PLM05	Percentage of switch-case in total of if-else and switch-case branches.
	PLM06	The ratio of the number of keyword struct to the number of SLOC.
	PLM07	The ratio of the number of keyword typedef to the number of SLOC.
	PLM08	The ratio of the number of keyword namespace to the number of SLOC.
	PLM09	The ratio of the number of keyword using to the number of SLOC.
	PLM10	The ratio of the number of keyword goto to the number of SLOC.
	PLM11	The ratio of the number of keyword const to the number of SLOC.
	PLM12	The ratio of the number of keyword template ^d to the number of SLOC.
	PLM13	The ratio of the number of keyword operator ^d to the number of SLOC.
	PLM14	The ratio of the number of keyword try ^d to the number of SLOC.
	PLM15	The ratio of the number of keyword catch ^d to the number of SLOC.

Table C.1 Continued

Dimension	Metric	Description
Programming Language Metrics (PLM)	PLM16	The ratio of the number of keyword throw ^d to the number of SLOC.
	PLM17	The ratio of the number of keyword explicit ^d to the number of SLOC.
	PLM18	The ratio of the number of keyword mutable ^d to the number of SLOC.
	PLM19	The ratio of the number of keyword volatile ^d to the number of SLOC.
	PLM20	The ratio of the number of casting operators ^{de} to the number of SLOC.
	PLM21	The ratio of the number of keyword auto ^{df} to the number of SLOC.
	PLM22	The ratio of the number of keyword constexpr ^{df} to the number of SLOC.
	PLM23	The ratio of the number of keyword nullptr ^{df} to the number of SLOC.
	PLM24	The ratio of the number of keyword static_assert ^{df} to the number of SLOC.
	PLM25	Whether the Rvalue reference ^{df} appears in the parameter passing.

^a SLOC (Source Lines of Code) is a count of lines in the text of source code excluding blank and pure comment lines.

^b<http://en.cppreference.com/w/c/keyword>

^c<http://en.cppreference.com/w/cpp/keyword>

^d Only exists in C++ but not C programming language.

^e The casting operators included in this study are: dynamic_cast, static_cast, const_cast and reinterpret_cast.

^f Since C++11.

Appendix D

Post-Experiment Questionnaire

Post-Experiment Questionnaire

Thank you very much for participating in our code readability survey (<http://gamicros.cs.cityu.edu.hk/>). We would like to ask you a few questions. Your support and cooperation would be greatly appreciated!

What is your age? _____

What is your gender? Male_____ Female_____

How many years of Java development experience do you have? _____

Have you been part of a similar survey before? Yes_____ No_____

To what extent do you disagree/agree with the following statements?

1 = Strongly Disagree

2 = Disagree

3 = Neither Agree nor Disagree

4 = Agree

5 = Strongly Agree

Perceived Ease of Use	1	2	3	4	5
Learning to use GamiCRS does not require a lot of mental effort.					
It is easy for me to become skillful at using GamiCRS.					
My interaction with GamiCRS is clear and understandable.					
Overall, I find GamiCRS easy to use.					
Perceived Usefulness	1	2	3	4	5
I would like to receive judgements from others on whether my source code is readable or not.					
GamiCRS helps me to better understand the importance of code readability.					
GamiCRS promotes me to write more readable code.					
Overall, I find GamiCRS useful.					
User Satisfaction	1	2	3	4	5
I find the user interfaces in GamiCRS are attractive.					
I have fun using GamiCRS.					
I have a generally favorable attitude toward using GamiCRS.					
Intention to Use	1	2	3	4	5
I would like to use GamiCRS once again.					

If you have points, badges, and leaderboards on your page, please answer the following questions. If not, please leave them blank.

Attitude Toward Game Design Elements	1	2	3	4	5
The point system in GamiCRS motivates me to participate more than usual.					
The badge system in GamiCRS motivates me to participate more than usual.					
The leaderboard system in GamiCRS motivates me to participate more than usual.					
Overall, I find the various game design elements in GamiCRS are well integrated.					

Any other feedback?

Thank you for completing this questionnaire!

Bibliography

- [1] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*, pages 235–241. IEEE, 2002.
- [2] Andrés Francisco Aparicio, Francisco Luis Gutiérrez Vela, José Luis González Sánchez, and José Luis Isla Montes. Analysis and application of gamification. In *Proceedings of the 13th International Conference on Interacción Persona-Ordenador*, page 17. ACM, 2012.
- [3] S. Arabyarmohamady, H. Moradi, and M. Asadpour. A coding style-based plagiarism detection. In *Proceedings of 2012 International Conference on Interactive Mobile and Computer Aided Learning (IMCL)*, number Imcl, pages 180–186. IEEE, nov 2012.
- [4] Ian Arawjo, Cheng-Yao Wang, Andrew C Myers, Erik Andersen, and François Guimbretière. Teaching programming with gamified semantics. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 4911–4923. ACM, 2017.

- [5] Pavel Avgustinov, Arthur I Baars, Anders S Henriksen, Greg Lavender, Galen Menzel, Oege De Moor, Max Schafer, and Julian Tibble. Tracking Static Analysis Violations over Time to Capture Developer Characteristics. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 437–447. IEEE, may 2015.
- [6] Titus Barik, Emerson Murphy-Hill, and Thomas Zimmermann. A perspective on blending programming environments and games: Beyond points, badges, and leaderboards. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, pages 134–142. IEEE, 2016.
- [7] Hamid Abdul Basit, Simon J Puglisi, William F Smyth, Andrew Turpin, and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering companion papers - ESEC-FSE companion '07*, number January, page 513, New York, New York, USA, 2007. ACM Press.
- [8] R E. Berry and B a.E. Meekings. A style analysis of C programs. *Communications of the ACM*, 28(1):80–88, jan 1985.
- [9] Benjamin Biegel, Fabian Beck, Benedikt Lesch, and Stephan Diehl. Code tagging as a social game. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 411–415. IEEE, 2014.
- [10] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. Extended models on the impact of identifier style on effort and

comprehension. Technical report, Technical Report LOY110720, Loyola University in Maryland, 2011.

- [11] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, apr 2013.
- [12] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 158–167. IEEE, may 2009.
- [13] Barry Boehm and Victor R Basili. Software defect reduction top 10 list. *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*, 34(1):75, 2007.
- [14] Jürgen Börstler, Michael E. Caspersen, and Marie Nordström. Beauty and the beast: on the readability of object-oriented example programs. *Software Quality Journal*, 24(2):231–246, Jun 2016.
- [15] Jurgen Borstler and Barbara Paech. The Role of Method Chains and Comments in Software Readability and Comprehension-An Experiment. *IEEE Transactions on Software Engineering*, 42(9):886–898, sep 2016.
- [16] Edgar Brunner and Ullrich Munzel. The nonparametric behrens-fisher problem: asymptotic theory and a small-sample approximation. *Biometrical journal*, 42(1):17–25, 2000.
- [17] Raymond P L Buse and Westley R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, jul 2010.

- [18] Raymond P L Buse and Thomas Zimmermann. Information needs for software development analytics. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 987–996. IEEE, jun 2012.
- [19] Raymond PL Buse, Caitlin Sadowski, and Westley Weimer. Benefits and barriers of user evaluation in software engineering research. *ACM SIGPLAN Notices*, 46(10):643–656, 2011.
- [20] Raymond PL Buse and Westley R Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130. ACM, 2008.
- [21] Alastair Carruthers and Jean Carruthers. Introduction. *Dermatologic Surgery*, 39(1pt2):149, jan 2013.
- [22] Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.
- [23] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, Trang Pham, Aditya Ghose, and Tim Menzies. A deep learning model for estimating story points. *arXiv cs.SE*, 9:00489, 2016.
- [24] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, 1993.
- [25] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.

- [26] Dennis Coon and John O Mitterer. *Introduction to psychology: Gateways to mind and behavior with concept maps and reviews*. Cengage Learning, 2012.
- [27] Lee J. Cronbach. Coefficient alpha and the internal structure of tests. *Psychometrika*, 16(3):297–334, sep 1951.
- [28] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 107–118, New York, New York, USA, 2015. ACM Press.
- [29] Hoa Khanh Dam, Truyen Tran, John Grundy, and Aditya Ghose. DeepSoft: a vision for a deep model of software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, volume 1691, pages 944–947, New York, New York, USA, feb 2016. ACM Press.
- [30] Fred D Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, pages 319–340, 1989.
- [31] Fred D Davis, Richard P Bagozzi, and Paul R Warshaw. User acceptance of computer technology: a comparison of two theoretical models. *Management science*, 35(8):982–1003, 1989.
- [32] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A Tutorial on the Cross-Entropy Method. *Annals of Operations Research*, 134(1):19–67, feb 2005.
- [33] Edward L Deci, Richard Koestner, and Richard M Ryan. Extrinsic rewards and intrinsic motivation in education: Reconsidered once again. *Review of educational research*, 71(1):1–27, 2001.

- [34] Edward L Deci and Richard M Ryan. *Handbook of self-determination research*. University Rochester Press, 2002.
- [35] Paul Denny. The effect of virtual achievements on student engagement. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 763–772. ACM, 2013.
- [36] Sebastian Deterding. Situated motivational affordances of game elements: A conceptual model. In *Gamification: Using game design elements in non-gaming contexts, a workshop at CHI*, 2011.
- [37] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. From game design elements to gamefulness: defining gamification. In *Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments*, pages 9–15. ACM, 2011.
- [38] Robert F DeVellis. *Scale development: Theory and applications*, volume 26. Sage publications, 2016.
- [39] Haibiao Ding and Mansur H. Samadzadeh. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software*, 72(1):49–57, jun 2004.
- [40] Anhai Doan, Raghu Ramakrishnan, and Alon Y. Halevy. Crowdsourcing systems on the World-Wide Web. *Communications of the ACM*, 54:86, apr 2011.
- [41] Jonathan Dorn. A General Software Readability Model. *University of Virginia, Charlottesville, Virginia*, pages 1–62, 2012.

- [42] Jonathan Dorn. A general software readability model. *MCS Thesis available from* (<http://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf>), 2012.
- [43] Theo Downes-Le Guin, Reg Baker, Joanne Mechling, and Erica Ruylea. Myths and realities of respondent engagement in online surveys. *International Journal of Market Research*, 54(5):1–21, 2012.
- [44] John V Farr. *Systems life cycle costing: economic analysis, estimation, and management*. CRC Press, 2011.
- [45] Rudolph Flesch. A new readability yardstick. *Journal of applied psychology*, 32(3):221, 1948.
- [46] Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 402–413, New York, New York, USA, 2014. ACM Press.
- [47] Wei Fu and Tim Menzies. Easy over hard: A case study on deep learning. *arXiv preprint arXiv:1703.00133*, 2017.
- [48] Robert Green and Henry Ledgard. Coding guidelines. *Communications of the ACM*, 54(12):57, dec 2011.
- [49] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 631–642, New York, New York, USA, aug 2016. ACM Press.

- [50] Juho Hamari, Jonna Koivisto, and Harri Sarsa. Does gamification work?—a literature review of empirical studies on gamification. In *System Sciences (HICSS), 2014 47th Hawaii International Conference on*, pages 3025–3034. IEEE, 2014.
- [51] Johannes Harms, Stefan Biegler, Christoph Wimmer, Karin Kappel, and Thomas Grechenig. Gamification of online surveys: Design process, case study, and evaluation. In *Human-Computer Interaction*, pages 219–236. Springer, 2015.
- [52] Johannes Harms, Dominik Seitz, Christoph Wimmer, Karin Kappel, and Thomas Grechenig. Low-cost gamification of online surveys: Improving the user experience through achievement badges. In *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*, pages 109–113. ACM, 2015.
- [53] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.
- [54] Joseph G. Ibrahim, Ming-Hui Chen, and Stuart R. Lipsitz. Monte Carlo EM for Missing Covariates in Parametric Regression Models. *Biometrics*, 55(2):591–596, jun 1999.
- [55] Barry Kantowitz, Henry Roediger III, and David Elmes. *Experimental psychology*. Nelson Education, 2014.
- [56] Karl M Kapp. *The gamification of learning and instruction: game-based methods and strategies for training and education*. John Wiley & Sons, 2012.
- [57] Phil Kim. *Convolutional Neural Network*, pages 121–147. Apress, Berkeley, CA, 2017.

- [58] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. pages 1–15, 2014.
- [59] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. Robust Statistical Methods for Empirical Software Engineering. *Empirical Software Engineering*, 22(2):579–630, apr 2017.
- [60] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012.
- [61] Ivan Krsul and Eugene H. Spafford. Authorship analysis: identifying the author of a program. *Computers & Security*, 16(3):233–257, jan 1997.
- [62] Janaki Kumar. Gamification at work: Designing engaging business software. In *International Conference of Design, User Experience, and Usability*, pages 528–537. Springer, 2013.
- [63] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481. IEEE, nov 2015.
- [64] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, dec 1989.
- [65] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

- [66] Taek Lee, Jung Been Lee, and Hoh Peter In. A study of different coding styles affecting code readability. *International Journal of Software Engineering and its Applications*, 7(5):413–422, 2013.
- [67] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.
- [68] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models. *Proceedings of the 30 th International Conference on Machine Learning*, 28:6, 2013.
- [69] G MacBeth, E Razumiejczyk, and R Ledsema. Cliff ’s Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10:545–555, 2012.
- [70] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, mar 1947.
- [71] Andrzej Marczewski. *Gamification: a simple introduction*. Andrzej Marczewski, 2013.
- [72] G Harry Mc Laughlin. Smog grading-a new readability formula. *Journal of reading*, 12(8):639–646, 1969.
- [73] Simon McCallum. Gamification and serious games for personalized health. *Stud Health Technol Inform*, 177(2012):85–96, 2012.

- [74] Qing Mi, Jacky Keung, Jianglin Huang, and Yan Xiao. Using eye tracking technology to analyze the impact of stylistic inconsistency on code readability. In *Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on*, pages 579–580. IEEE, 2017.
- [75] Qing Mi, Jacky Keung, and Yang Yu. Measuring the Stylistic Inconsistency in Software Projects using Hierarchical Agglomerative Clustering. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE 2016*, pages 1–10, New York, New York, USA, 2016. ACM Press.
- [76] Richard J. Miara, Joyce a. Musselman, Juan a. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, 26(11):861–867, nov 1983.
- [77] Tomas Mikolov, Greg Corrado, Kai Chen, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, pages 1–12, 2013.
- [78] Roberto Minelli, Andrea Mocci, and Michele Lanza. Free hugs—praising developers for their actions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 555–558. IEEE, 2015.
- [79] Sebastian C. Muller and Thomas Fritz. Stuck and Frustrated or in Flow and Happy: Sensing Developers’ Emotions and Progress. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 688–699. IEEE, may 2015.

- [80] Cristina Ioana Muntean. Raising engagement in e-learning through gamification. In *Proc. 6th International Conference on Virtual Learning ICVL*, number 42, pages 323–329, 2011.
- [81] Karin Neubert and Edgar Brunner. A studentized permutation test for the non-parametric Behrens-Fisher problem. *Computational Statistics & Data Analysis*, 51(10):5192–5204, jun 2007.
- [82] Markus Neuhauser. *Nonparametric statistical tests: A computational approach*. CRC Press, 2011.
- [83] Anh Tuan Nguyen and Tien N Nguyen. Graph-Based Statistical Language Model for Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume vol.1, pages 858–868. IEEE, may 2015.
- [84] Scott Nicholson. A recipe for meaningful gamification. In *Gamification in education and business*, pages 1–20. Springer, 2015.
- [85] Donald A Norman. *The design of everyday things: Revised and expanded edition*. Basic books, 2013.
- [86] Paul W Oman and Curtis R Cook. A paradigm for programming style research. *ACM SIGPLAN Notices*, 23(12):69–78, dec 1988.
- [87] Paul W. Oman and Curtis R. Cook. A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation - CSC '90*, pages 244–250, New York, New York, USA, 1990. ACM Press.
- [88] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *Proceedings of the 8th International*

- Conference on Knowledge Science, Engineering and Management - Volume 9403*, KSEM 2015, pages 547–553, New York, NY, USA, 2015. Springer-Verlag New York, Inc.
- [89] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, volume 11, page 73, New York, New York, USA, 2011. ACM Press.
- [90] Michael J. Rees. Automatic assessment aids for pascal programs. *SIGPLAN Not.*, 17(10):33–42, October 1982.
- [91] Paige Rodeghero, Cheng Liu, Paul W. McBurney, and Collin McMillan. An Eye-Tracking Study of Java Programmers and Application to Source Code Summarization. *IEEE Transactions on Software Engineering*, 41(11):1038–1054, nov 2015.
- [92] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen’s d for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.
- [93] Bill Rounagas and Fabiano Dalpiaz. A model-driven framework for educational game design. In *International Conference on Games and Learning Alliance*, pages 1–11. Springer, 2015.
- [94] Richard M Ryan and Edward L Deci. Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary educational psychology*, 25(1):54–67, 2000.

- [95] Richard M Ryan and Edward L Deci. Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being. *American psychologist*, 55(1):68, 2000.
- [96] Adrian Vicentiu Salcu and Carmen Acatrinei. Gamification applied in affiliate marketing. case study of 2parale. *Management & Marketing*, 8(4):767, 2013.
- [97] Yui Sasaki, Yoshiki Higo, and Shinji Kusumoto. Reordering Program Statements for Improving Readability. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 361–364. IEEE, mar 2013.
- [98] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 417–427, 2017.
- [99] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, volume 2016-July, pages 1–10. IEEE, may 2016.
- [100] Katie Seaborn and Deborah I Fels. Gamification in theory and action: A survey. *International Journal of Human-Computer Studies*, 74:14–31, 2015.
- [101] Bonita Sharif and Jonathan I. Maletic. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 196–205. IEEE, jun 2010.

- [102] Leif Singer and Kurt Schneider. It was a bit of a race: Gamification of version control. In *Games and Software Engineering (GAS), 2012 2nd International Workshop on*, pages 5–8. IEEE, 2012.
- [103] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. Improving developer participation rates in surveys. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, pages 89–92. IEEE, 2013.
- [104] Remco Snijders, Fabiano Dalpiaz, Sjaak Brinkkemper, Mahmood Hosseini, Raian Ali, and Atilla Ozum. Refine: A gamified platform for participatory requirements engineering. In *Crowd-Based Requirements Engineering (CrowdRE), 2015 IEEE 1st International Workshop on*, pages 1–6. IEEE, 2015.
- [105] Zephyrin Soh, Zohreh Sharafi, Bertrand Van den Plas, Gerardo Cepeda Porras, Yann-Gael Gueheneuc, and Giuliano Antoniol. Professional status and expertise for UML class diagram comprehension: An empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 163–172. IEEE, jun 2012.
- [106] Christian Szegedy. Scene classification with inception-7.
- [107] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9. IEEE, jun 2015.
- [108] Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi, and Maryan Yatim. Impact of Programming Features on Code Readability. *International Journal of Software Engineering and Its Applications*, 7(6):441–458, nov 2013.

- [109] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, volume 14-22-May-, pages 297–308, New York, New York, USA, 2016. ACM Press.
- [110] Xiaoran Wang, Lori Pollock, and K. Vijay-Shanker. Automatic segmentation of method code into meaningful blocks: Design and evaluation. *Journal of software: Evolution and Process*, 26(1):27–49, 2014.
- [111] Kevin Werbach and Dan Hunter. *For the win: How game thinking can revolutionize your business*. Wharton Digital Press, 2012.
- [112] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 87–98, New York, New York, USA, 2016. ACM Press.
- [113] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE, may 2015.
- [114] Darren Wilson, Cynthia Calongne, and S Brook Henderson. Gamification challenges and a case study in online learning. *Internet Learning*, 4(2):8, 2015.
- [115] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering*, 5589(c):1–1, 2017.

- [116] Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Educational software engineering: Where software engineering, education, and gaming meet. In *Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change*, pages 36–39. IEEE Press, 2013.
- [117] Ye Zhang and Byron Wallace. A Sensitivity Analysis of (and Practitioners’ Guide to) Convolutional Neural Networks for Sentence Classification. oct 2015.
- [118] Gabe Zichermann and Christopher Cunningham. *Gamification by design: Implementing game mechanics in web and mobile apps*. O’Reilly Media, Inc., 2011.