Impact of Programming Features on Code Readability

Y. Tashtoush

Z. Odat

Izzat M. Alsmadi
Texas A&M University-San Antonio, ialsmadi@tamusa.edu

M. Yatim

International Journal of Software Engineering and Its Applications
Vol.7, No.6 (2013), pp.441-458
http://dx.doi.org/10.14257/ijseia.2013.7.6.38

Impact of Programming Features on Code Readability

Yahya Tashtoush[1], Zeinab Odat[2], Izzat Alsmadi[3] and Maryan Yatim[4]
[1, 2, 4]JUST Uni.
[3]Yarmouk Uni.
[1]yahyat@just.edu.jo
[3]ialsmadi@yu.edu.jo

Abstract

Readability is one important quality attributes for software source codes. Readability has also significant relation or impact with other quality attributes such as: reusability, maintainability, reliability, complexity, and portability metrics. This research develops a novel approach called Impact of Programming Features on Code Readability (IPFCR), to examine the influence of various programming features and the effect of these features on code readability. A code Readability Tool (CRT) is developed to evaluate the IPFCR readability features or attributes.

In order to assess the level if impact that each one of the 25 proposed readability features may have, positively or negatively on the overall code readability, a survey was distributed to a random number of expert programmers. These experts evaluated the effect of each feature on code readability, based on their knowledge or experience. Expert programmers have evaluated readability features to be ordered then classified into positive and negative factors based on their impact on code readability or understanding. The survey responses were analyzed using SPSS statistical tool. Most of proposed code features showed to have significantly positive impact on enhancing readability including: meaningful names, consistency, and comments. On the other hand, fewer features such as arithmetic formulas, nested loops, and recursive functions showed to have a negative impact. Finally, few features showed to have neutral impact on readability.

Keywords: *Code readability; Software quality; Reusability, Maintainability*

1. Introduction

Code Readability can be defined as a human judgment on how easy it is to understand a program source code [1], the ratio between lines of code and number of commented lines [2], writing to people not to computers [3], making a code locally understandable without searching for declarations and definitions [4], and also, the average number of right answers to a series of questions about a program in a given length of time [5]. Definitions show clearly that such quality attribute is related to some other attributes such as: understandability, usability, reusability, complexity, or maintainability.

Although readable code is less erroneous [3], more reusable [3, 6], easier to maintain [7], quicker to modify [8, 9], and more consistent [3], code readability is not easy to measure by a deterministic function same as maintainability, reliability, and reusability. Moreover, a source code may be considered a readable one for a programmer but it might not be for another, and that was the inspiration of this research which examines readability through studying several code features that affect readers' understanding of an existing source code. Studying the Impact of Programming Features on Code Readability

The main goal this paper is to develop a tool that can automatically collect different readability attributes. We then want to see the value of those low level readability attributes and their ability or level of impact on the overall readability.

Nowadays, most software codes are written by distant teams. Developers may frequently join or leave software companies. It's very important for source codes to be understandable so it can be easily adapted. Such codes are library functions and packages in programming languages such as: C++, C#, and Java. Each programmer may want to review code and adapt it according to his/her own needs; so readability is needed. Furthermore, the ability of reading and understanding a program written by others is a critical job. Software programming companies depend on team work instead of one programmer, so each one will write a fragment as part of a team effort. In addition, tailoring a ready code would be very difficult to specify tasks then modify them (*e.g.*, some programs may have thousands of lines of code such as network protocols).

Programming courses are taught through various techniques and exercises introduced to students to ensure their skills. Thus, instructors should insure that these exercises are given clear and understandable to students, in order to get clear answers. On the other hand, students should submit clear and readable answers in order for instructors to review and correct. Moreover, programmers may forget variables purposes when using meaningless names, or forget functional purposes when omitting comments. Thus, using unreadable program codes may lead to code misunderstanding.

Software quality is a comprehensive characteristic of the whole software life cycle, defined as what software product should be and what it must contain. The quality of software is defined by several characteristics such as: software maintainability, reliability, reusability, testability, and readability. Literature reviews have referred to maintainability and testability as the main quality factors [3], also, readability has always been the reason behind maintainable code, and, software quality has to be improved in all dimensions; so code readability is a priority.

The rest of the paper is organized as follows. Section 2 presents readability and related work. Section 3 discusses the Impact of Programming Features on Code Readability (IPFCR) approach and how it works. Section 4 presents the methodology and experiment of the approach. Section 5 improves the CRT tool and represents the results of the research. Conclusion of the research is given in section 6. Finally, Section 7 presents the future work.

2. Documentation and Code Readability

Readability in software products contains documentation and source code readability.

2.1. Documentation Readability

Documentation readability means a detailed system description which will be shared between software engineering team and customer, to hold proper understanding for the system; it plays a key role especially in large systems. Many requirements are associated in system documentation [10, 11]; it should be an information repository system to be used by maintenance engineers, it should provide information for managers to help them in planning, it also should contain the budget and schedule for software development process, and finally it should tell users how to use and administer the system.

The software documentation must be understandable for programmer and customer who are not professionals. In agile software, developers focus on working software over comprehensive documentation with customer collaboration and contract negotiation, in other words it means that documentation must be readable in all versions to give customers the ability for understanding software specifications and so adapting it.

Unreadable documentation may cause documentation errors and omissions which can lead to errors by end-users and consequent system failures which will complicate using or maintenance activities. Although some studies are concerned with source code readability, they neglected documentation readability (*e.g.*, [6]). This is because readability in both cases takes different features to be measured. While both documentation and code may share many common readability features, some aspects maybe unique for each one of them.

2.2. Source Code Readability

Code readability is usually connected with comments and naming standards. While those are two major factors that impact readability, there are also some other aspects to consider.

One of the new concepts that appear in relation with software quality and readability is the refactoring process. It means changing code structure externally without affecting internal behavior to improve readability, flexibility, and enable easier modifications [12-14]. Refactoring often aims to add new non functional objectives for code, without affecting its main purpose by testing the code frequently.

In [15] the author defines "improving code readability" as the first advantage for code refactoring. The survey briefly explains code refactoring techniques such as: field encapsulation, type generalization, or methods renaming. Refactoring advantages include: improving readability, maintainability, less complexity, and reusability.

3. Related Work

The main title (on the first page) should begin 1 3/16 inches (7 picas) from the top edge of the page, centered, and in Times New Roman 14-point, boldface type. Capitalize the first letter of nouns, pronouns, verbs, adjectives, and adverbs; do not capitalize articles, coordinate conjunctions, or prepositions (unless the title begins with such a word). Please initially capitalize only the first word in other titles, including section titles and first, second, and third-order headings (for example, "Titles and headings" — as in these guidelines). Leave two blank lines after the title.

Readability research is not only used for code readability. Documents, web-pages, or any form of text may be subjective to such evaluation or tests. Hence, there are some readability metrics that are used to evaluate natural language in general regardless of the purpose that the language is used for.

There are then some popular metrics or tests to evaluate words or statements' readability. The most popular tests are Flesch Reading Ease and the Flesch-Kincaid [16]. The readability grading level depends on several factors such as: word length, sentence length, word form, and syllables or letters. They state that shortening the sentences and words will make it easier for reader to understand the text if compared with long statements. A new and improved model for readability is used in [17], that includes checking code readability with the usage of previous text readability test. In the Flesch Reading Ease test (FRES), high scores indicate a document that is easier to read. Lower scores indicate a document that is more difficult to read. The formula for (FRES) test is:

EQ. 1: R.E: = 206.835 - (0.846 *wl) - (1.015 * sl)

Where:
R.E. = Reading Ease
wl = Word Length
sl = Average Sentence Length

The model uses characters instead of alphabets, keywords and identifiers are used instead of words, blocks are equivalent to paragraphs, statements are equivalent to sentences, and modules are equivalent to chapters.

In 1997, Chung Yung proposed a model that integrates software science metrics with code complexity, respecting the readability of the algorithm implemented [18]. For the program source code, author proposed four metrics which are; the number of unique operators, the number of unique operands, the total number of operator occurrences, and the total number of operand occurrences. The operands are the variables or constants, and the operators are the symbols or combinations of symbols that affect the values or ordering of operands [19]. The motivation is the correlation between software readability and maintainability.

Buse and Weimer in “Learning a Metric for Code Readability” [1] constructed an automated readability tool. They investigated human annotators to judge selected snippets (short codes) of Java code, and then compared results with their measure. Their tool was able to achieve 80% readability prediction accuracy. Correlation between readability and two other software quality metrics was studied, which are: Bugs or faults detection and code evolution or changes. Choosing of fine grained codes neglects the effect of code volume. They used snippets and studied code features line by line. In this vision, code volume is the first feature that has a direct effect on readability; large codes will not be easy as small ones; 5 or 6 lines for example. Readability features collected from literature studies depend on coding style and complexity.

A new de-compilation mechanism was proposed to achieve high readable decompiled source codes [20]. Decompilation is the process of transferring binary code into high level source codes such as IDA Hex rays, Boomerang, and Dcc. In all these tools, poor readability was found via low accuracy in identifying variables, functions, and structures. Most challenges were related to: information recovery and flow graph building, especially for loops and condition statements. Their new algorithm focused on challenges to produce a high readable code. The model was able to achieve a high readable source code by identifying parameters and variables, removing redundant variables, analyzing data dependency and flow, and extract structures.

Cowan proposed a new approach enhancing code readability to benefit reengineering technologies, which are using SGML (Standard Generalized Markup Language), to embed semantic and syntax with program code, which uses code visualization and text database [21]. The goal of code visualization is to connect the mental image of code writer to the mental image of program reader [22].

The researcher in “Improving Software Quality through the Development of Code Readability” [23] developed a new and effective automated readability measure which can calculate the readability and complexity of the software better than human judgments. The researcher gathered a number of snippets from open source codes over the internet, and relied on professional programmers to judge these snippets and to value their complexity based on some given features like keywords, comments, loops, lines, *etc*.

Philip A. Relf [24] implemented an empirical study on a group of programmers – students and experts – to study the effect of naming styles on source code readability and maintainability. The study investigated and collected nineteen rules for meaningful identifier naming styles, and produced a Java code editor which has the ability to check identifier names.

Using of meaningful names has been also studied in [25] and authors tried to correlate their results with the tool: FinBugs, software quality metrics tool for Java codes. Authors expanded their research by studying more relations with more quality metrics. They conducted research on fine grained parts, which are Java methods [26]. They use identifiers naming guidelines with some adoption; Buse and Weimers' readability metrics [1] as readability method, and Cyclomatic complexity as controversial metric. They implement previous metrics on several Java open source codes, and then claim that poor quality identifiers names cause a less readable and less maintainable source code, and thus poor quality software.

In [27], the researchers developed a new and automatic system to add blank lines into source code in order to improve code readability and locate points for internal documentation. They developed a tool, SEGMENT, which "inputs a Java method, and outputs a segmented version that separates meaningful blocks by blank line insertions. Not only can the output segmentation help in readability, it can provide hints for where to place internal comments." The system evaluation showed that the automatic inserted blank lines are as good as the blank lines inserted by programmers and developers. Moreover, the evaluation showed that the system uses vertical spacing in places where programmers think it is better to use.

4. IPFCR Approach

Author names and affiliations are to be centered beneath the title and printed in Times New Roman 12-point, non-boldface type. Multiple authors may be shown in a two or three-column format, with their affiliations below their respective names. Affiliations are centered below each author name, italicized, not bold. Include e-mail addresses if possible. Follow the author information by two blank lines before main text.

We tried to extract several readability related features from the source code that can be automatically collected.

The main concern of The Impact of Programming Features on Code Readability (IPFCR) approach is to generate exact readable values for Java codes by defining their features. A survey was distributed to gather programmers' satisfaction of each readable feature through scaling their assessments from (1-5). These evaluations were analyzed to create the weighting factors for our readability metrics' tool, called Code Readability Tool (CRT). Figure 1 shows the workflow of the IPFCR approach.

The IPFCR workflow starts by gathering the survey responses and analyzing them. Users or experts' surveys or opinions are used to weight the readability features. The readability features are specified through parsing Java codes by the CRT tool. For each feature there's a dedicated formula that uses previous features weights (values). The summation of all those formulae values represents then the readability Value.

Meaningful Names for Classes, Methods, Variables, etc.

One of the importance issues that are usually tackled in readability evaluation is the selection of names for program components: Packages, classes, methods, variables, *etc*. Coding standards and naming conventions use some rules for names in code components. However and in order to judge whether those names are (meaningful) or not, which is something partially subjective, users may give conflicting opinions on those names. This process can be further complex if a tool needs to automatically judge such readability issue. In general, there are some basic assessment issues that can be easily evaluated (*e.g.*, name should not be less than 3 letters). However, some more subjectively aspects related to judging whether the name is meaningful or not can be hard to automate or detect by a tool. For example, the general meaning of "meaningful names" is that names should reflect purpose of using such code component. In order to evaluate the fitness or relevancy of the name to the scope, natural language processing maybe used to study the relation between names in the same component or containers. Such issue may also be challenged by the fact the components' names may contain parts of more than one word connected together in a special way (*e.g.*, calAveValues). The example shows three words connected together, abbreviated in a special way that may not be understood by a parser or language processor that looks for words in the dictionary. The correct divisions of methods and their responsibilities may also share to this issue. For example, cohesion is a design quality aspect that stresses that a component should contain only inner related items. As such method names for example should not include (and) which may indicate mixing more than one function together that should be separated.

As such, meaningful names evaluation applied in the algorithms below depends only on the number of letters' count. "Choosing good names takes time but saves more than it takes ". This is a quote from a very good book that talks about code readability (Clean code by Robert Martin, [31]).

The features that were proposed, developed and collected will be explained in the next paragraphs. In each formula, the weight factor is the one that is averaged as the output from experts' assessment.

- **AVMN**

EQ. 2: AVMN = (VMN / V) * MNW

Where:
AVMN: Average Variables with Meaningful Names
VMN: Number of Variables with Meaningful Names
V: Number of Variables
MNW: Meaningful Names Weight

- **AFMN: Average Function with Meaningful Names**

EQ. 3: AFMN = (FMN / F) * MNW

Where:
AFMN: Average Function with Meaningful Names
FMN: Number of Functions with Meaningful Names
F: Number of Functions
MNW: Meaningful Names Weight

- **WLC: White lines per code line**

EQ. 4: WLC = (BL1 / BL2) * SW

Where:
WLC: White lines per code line
BL1: Number of Places which have a Blank Line
BL2: Number of Places which Blank lines are necessary

SW: Spacing Weight

- **Indents**

EQ. 5: $I = (I1/ I2) * IW$

Where:
I: Indents
I1: Number of Places which have Indents
I2: Number of Places which Indents are needed
IW: Indent Weight

- **CM**

EQ. 6: $Cm = (Cml/CL) + (Cml / 0.2 * CW)$

Where:
Cm: Comments
Cml: Number of Comments Lines
CL: Number of Code lines
CW: Comment Weight

- **Scope**

EQ. 7: $S = (MSV / MaxSV) * SpW$

Where:
S: Scope
MSV: Median Scopes Volumes
MaxSV: Maximum Scopes Volume
SpW: Scopes Weight

- **Lines Length**

EQ. 8: $LL = (MLL / MaxLL) * LLW$

Where:
LL: Line Length
MLL: Median Lines Length
MaxLL: Maximum Line Length
LLW: Line Length Weight

- **Arithmetic Formulas**

EQ. 9: $ArF = e ^\wedge - (Fn / CL) * FW$

Where:
ArF: Arithmetic Formulas
Fn: Number of Formulae

CL: Number of Code Lines
FW: Formulas Weight

- **Average of If-else**

EQ. 10: AvIe = (IeS / CL) * IeW

Where:
AvIe: Average If-else
IeS: Number of “If” Statements
CL: Number of Code Lines
IeW: If else Weight

- **Nested If**

EQ. 11: Ni = e ^ (-MaxNiD) * NiW

Where:
Ni: Nested If
MaxNiD: Maximum Nested If Depth
NiW: Nested If Weight

- **Average For loop**

EQ. 12: AvFL = (FL/ L) * FLW

Where:
AvFL: Average For loop Between Other Types
FL: Number of "For" Loop
L: Number of Other Loops
FLW: For Loop Weight

- **Nested Loop**

EQ. 13: (NL) = e ^ (- MaxNLD) * NLW

Where:
NL: Nested Loop
MaxNLD: Maximum Nested Loop Depth
NLW: Nested Loop Weight

- **Recursive Functions**

EQ. 14: RF = e ^ (- R) * RW

Where:
RF: Recursive Functions
R: Number of Recursive
RW: Recursive Weight

- **Arrays**

EQ. 15: $Ar = (A / Is) * AW$

Where:
Ar:Arrays
A: Number of Arrays
Is: Number of Identifiers
AW: Arrays Weight

- **Class Distribution**

EQ. 16: $CD) = (MCV / MaxCV) * CW$

Where:
CD: Class Distribution
MCV: Median Classes Volume
MaxCV: Maximum Class Volume
CW: Class Weight

- **Inheritance**

EQ. 17: $In = (InC / C) * InW$

Where:
In: Inheritance
InC: Number of Inherited Classes
C: Number of Classes
InW: Inheritance Weight

- **Overriding**

EQ. 18: $Or = (PVFD/ FD) * OrW$

Where:
Or: Overriding
PVFD: Number of functions inherited from Pure Virtual Functions Declarations
FD: All Function Declarations
OrW: Overriding Weight

- **Consistency**

EQ. 19: $Cn = (Com / LSc) * CnW$

Where:
Cn: Consistency
Com: Number of Commas
LSc: Number of Lines in the source code
CnW: Consistency Weight

After analyzing the survey, readability features are evaluated to get a crisp feature weight.

The survey is created online. We built a list of experts searching through the Internet and contacting candidate experts until we have confirmed the participation of a significant number of experts located in different companies, countries or programming areas and languages.

The SPSS statistics tool is used to analyze the survey responses and produce the features' weight. CRT parses a Java code to specify its features, and then a specific formula is applied for each feature using weights to calculate the final readability value.

SPSS statistical set of tests presented efficient responses analysis. It has been assumed that personal information has a notable effect on people's evaluation; therefore it was included in the survey. The most useful information value was produced by integrating respondents' experiences with their jobs stated as (F measure). Based on this value, readability features were classified into positively impacting factors and negatively impacting ones in relation with feature impact on readability in general.

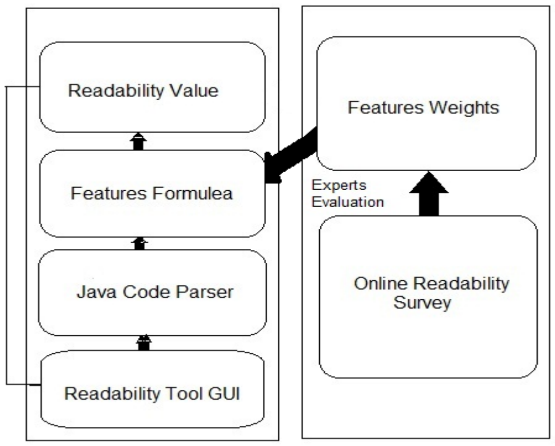

Figure 1. IPFCR Approach Workflow

5. Methodology

This research aims to check readability as a restricted value to be a useful indicator for code quality. This is accomplished by evaluating low level readability features in the source code then calculating how much each feature affects the readability quality factor based on a mixture of formulas collected from the source code and weights on each feature formula provided by experts.

Experts' Based Metrics' Weighting

There is a need to evaluate the importance of each one of the 22 collected readability related attributes on readability metric. Since readability is highly subjective and is judged by experts or users, we decided to give weights to the different collected attributes based on experts' judgments. A survey is then conducted that include questions related to the twenty two attributes and experts' opinions on the impact of such attributes on code readability. The survey responses of 141 random annotators at several software developing companies are assembled. A web based survey or questionnaire is built and requests were sent to candidates based on their expert and skills in the field of programming or software engineering.

Readability features are gathered from literature researches. Then the study forms an online questionnaire aims to evaluate the effect of each readability feature by getting responses from professional people depending on their years of experience in a specific job related to programming. Twenty three questions were asked in the feature survey, one for each feature.

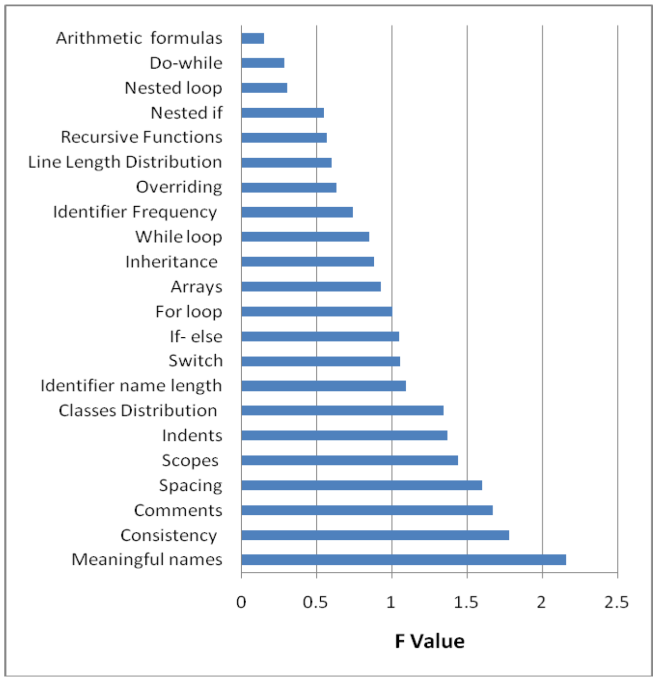

Figure 2. ANOVA Test (F) Measure for Factors with Experience and Job

Respondent quantifies feature effect, to be the feature weight, using result analysis and evaluation scale. Survey results the analysis using SPSS tool with several statistical analysis tests such as: frequency, reliability, descriptive statistics, and ANOVA test.

To weight each feature and to avoid any possible bias in one or more experts, the research uses ANOVA comparative test between questions and job experience. These two factors have a large effect on peoples' evaluations as mentioned in Figure 2, so we used a value (F) from ANOVA test to weight factors. (F) Is the degree of homogeneity between mean values for each question, and mean values of job experience, and it is expressed by the following equation:

$$\textbf{EQ. 20:} \quad F = BSSM / WSSM$$

Where:

BSSM: Mean Square Between values in the feature

WSSM: Mean Square Within values in the feature

Merging the impact of job experience with each factor evaluation is accomplished by using ANOVA test. Meaningful names, consistency, code lines, and comments are shown to have a positive impact on whole readability value. On the other hand, arithmetic formulas, nested loops, and recursive functions, have different impact (F measure) values, which mean negative impact. For each feature, the research develops a formula to check features' effect on code readability value using the previous weights.

IPFCR approach, used in this research, is built on readability features collected from programming studies in the literatures. The tool begins by searching for code features, and depends on factors' formula with defined weights. Then features effects will be a deterministic value. Table 1, shows the features used in this research with brief formulae and how the feature effect is calculated from the code by its average, maximum, and median.

6. CRT and Results

Readability Tool (CRT) is implemented using C# language. It evaluates readability of Java codes to develop a new readability metric based on collected features. The tool has the following operations: Import Java file, analyze parsed code, scopes and functions details, generate detail report, and update with weight values.

Table 1. Readability Features Formulae

	Factor	Average	Median	Maximum
1	Meaningful Names	*		
2	Comments	*		
3	Spacing			
4	Indents			
5	Short Scopes		*	*
6	Line Length distribution		*	*
7	Identifier name length		*	*
8	Arithmetic formulas	*		
9	Identifier frequency		*	*
10	If-else	*		
11	Nested if			*
12	Switch	*		
13	For loop	* of loops		
14	While loop	*of loops		
15	Do – while	*of loops		
16	Nested loop			*
17	Recursive			*
18	Arrays	* of identifiers		
19	Classes distribution		*	*
20	Inheritance			*
21	Overriding			*
22	Consistency	*		

CRT is tested by trying the usage of most system functions to ensure correct behavior and test the use cases of the tool. During first testing phase, three different Java codes were tested, taken form Java code websites. Several versions of each code were used to focus on one or two features. Note that each version was compared with its previous. The first code used was an implementation for Dijkstra shortest path algorithm with 80 lines of code, taken from [28]. The comparison of Dijikstra code versions is shown in Figure 4.

Figure 3 shows that the maximum value of readability was recorded when the nested-if-statement was removed. Afterwards the value started to decrease until it reached the minimum score when unbalanced classes were examined.

Features dependencies cause various changes in other values, for example, using meaningful names leads to longer name length, also the use of while-loop instead of for-loop affects scopes value and code lines. Moreover, noticed that using overriding did not affect the readability value taken from previous version, because other features have also been changed.

The second code was (SudokuPuzzel.Java); implementation of Sudoku puzzle game with code volume of 100 lines, taken from [29], its readability values are shown in Figure 4 which shows that the original code has a recursive call with the least readability value, but readability has increased to maximum by omitting its use. The use of comments, spacing, meaningful names and overriding raised the code readability result; on the other hand, class distribution feature had a negative effect by decreasing the value

The third code was date and time utility program with 112 code lines [30]. This program demonstrates the common use of Date functionality in Java programming scenarios. Several code versions have been studied to produce values in Figure 5 which shows that the original code has the least readability value, which begins continuously increasing with adding comments, meaningful names, indents and spacing, inheritance, and overriding.

As noticed in the 3 examples, the use of unbalanced classes decreases the code readability value. Features with positive impact on code readability contribute in raising the CRT value, unlike those with a negative impact. Second testing phase was studying fifty Java codes in order to examine the changes in readability based on each code feature. The 50 Java code sample approved CRT assumption which deals with object oriented technique. The code should contain almost all features in order to get an accurate readability value, the code with 42 lines scored the maximum readability value as shown in Figure 6. It is obvious in Figure 7 that readability value is not clearly decreasing by Line of Code (LOC) feature which approves its small impact.

But, readability value is clearly increasing by meanin 9gful and comments values for 50 Java sample Codes. Their positive impact was approved by ordering readability values of those factors' values as Figures 7 and 8 respectively. On the other hand, ordering readability values by nested loop and recursive factors shows their negative impact which decreases the readability value. So the negative impact was approved for those factors use, as shown in Figures 9 and 10 respectively.

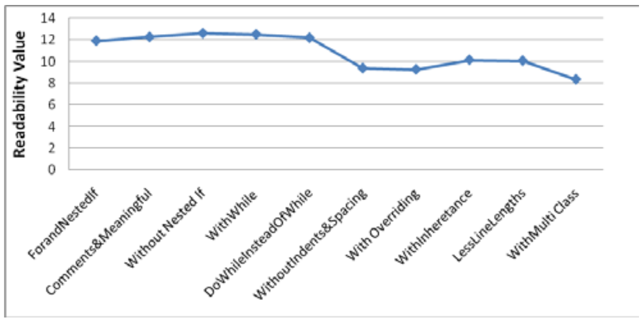

Figure 3. (Dijkstra.Java) Readability Values

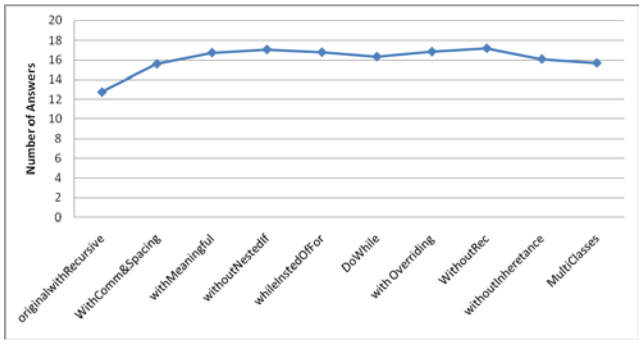

Figure 4. (Sudoku.Java) Readability Values

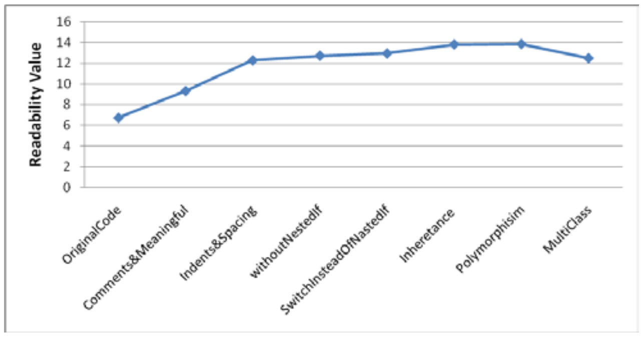

Figure 5. (DateAndTimeUtility.Java) Readability Values

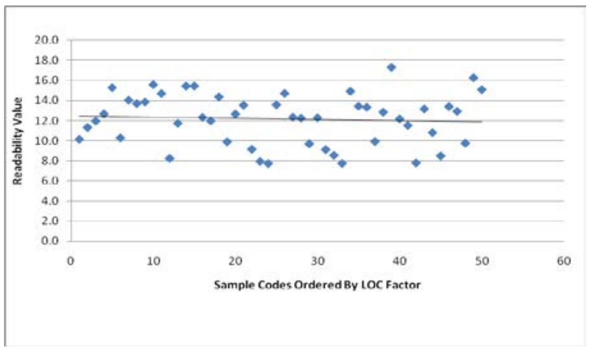

Figure 6. CRT Value for 50 Java Codes Ordered by LOC

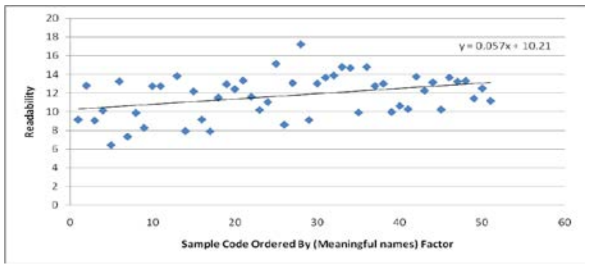

Figure 7. CRT Value for Codes Sample Ordered by (Meaningful Names)

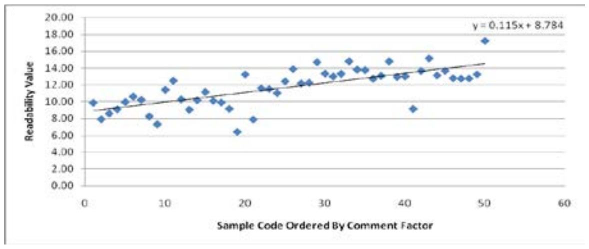

Figure 8. CRT Value for Codes Sample Ordered by (Comments) Factor

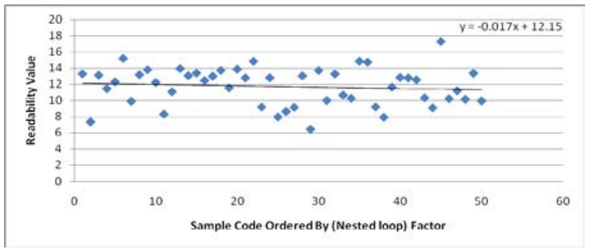

Figure 9. CRT Value for Codes Sample Ordered by (Nested Loop) Factor

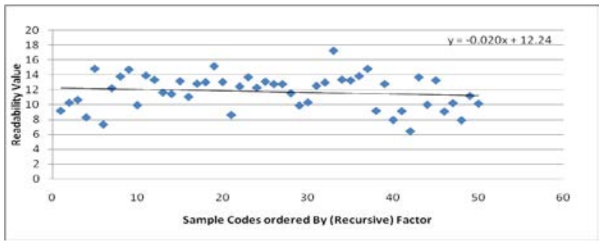

Figure 10. CRT Value for Codes Sample Ordered by (Recursive) Factor

7. Conclusion and Future Work

In this paper, a tool is developed to parse several low level readability features automatically from software source codes. For each one of the collected feature, a weight of that feature to express its level of impact on readability in general is added to the feature formula to be adjusted by human experts. Those programming human experts were asked to express their opinion throw a nominal level on the level of impact each feature may have an impact on code readability.

The survey responses of 141 random annotators or experts at several software developing companies were analyzed by the SPSS tool. Experts are selected to vary in country or location, nature of Software Company, age, *etc.*, to offset any possible bias in results. Readability features were weighted using ANOVA comparative test.

Features of high positive impact in improving readability include: meaningful names, comments, and consistency. Whereas, recursive functions, nested loops and arithmetic formulas were found to be of a negative impact on the general readability attribute. Others, such as, short scopes, identifier name length, identifier frequency, inheritance, overriding, if-else statement, switch statement, loops (for, while, do-while), and array were found to have no major effect on readability.

Future extensions of this work should extend the readability features to include more semantic aspects that may have better indication of code readability despite acknowledging the fact that such features can be complex to formulate and collect automatically by tools from the source code.

References

[1] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability", Software Engineering, IEEE Transactions, doi:10.1109/tse.2009.70, vol. 36, no. 4, **(2010)**, pp. 546-558.
[2] E. Collar and R. Valerdi, "Role of software readability on software development cost", Proceedings of the 21st Forum on COCOMO and Software Cost Modeling, Herndon, VA, **(2006)** October.
[3] R. Fitzpatrick, "Software quality: definitions and strategic issues", Reports, **(1996)**, pp. 1.
[4] K. Tokuno and S. Yamada, "Markovian Software Reliability Measurement with a Geometrically Decreasing Perfect Debugging Rate", IJMTM, **(2003)**, pp. 71-80.
[5] R. Land, "Measurements of software maintainability", Proceedings of ARTES Graduate Student Conference, ARTES, **(2002)**, pp. 1-7.
[6] A. Holliday, "Coding Format and Style", handout contains guidelines for coding format, **(2008)**.
[7] J. Prothero, "Usability Best Practices", White Paper from JackBe Corporation, **(2006)**, pp. 20-26.
[8] T. Tenny, "Program readability: Procedures versus comments", Software Engineering, IEEE Transactions on, doi>10.1109/32.6171, vol. 14, no. 9, **(1988)**, pp. 1271-1279.
[9] C. A. Cunha, J. L. Sobral and M. P. Monteiro, "Reusable aspect-oriented implementations of concurrency patterns and mechanisms", Proceedings of the 5th international conference on Aspect-oriented software development, **(2006)**, pp. 134-145.
[10] I. Sommerville, http://powershow.com/view/946f1-zM1O/Software_Documentation_Written_By_Ian_Sommerville_flash_ppt_presentation. [Accessed 07 December 12], **(2001)**.
[11] A. Mobasseri, "Improving Readability", 06-csrs2008-ArminMobasseri.pdf, **(2008)**.
[12] J. Viljamaa, "Refactoring I - Basics and Motivation", Helsinki, Seminar on Programming Paradigms, **(2000)**.
[13] E. Murphy-Hill and A. Black, "Breaking the barriers to successful refactoring", Software Engineering, ICSE'08. ACM/IEEE 30th International Conference, **(2008)**, pp. 421-430.
[14] S. Counsell and E. Nasseri, "System Evolution at the Attribute Level: an Empirical Study of Three Java OSS and their Refactorings", Journal of Computing and Information Technology, vol. 18, no. 2, **(2010)**.
[15] S. Kansal, "Refactor Code: A Review", IJCSt, **(2011)**, pp. 2-4.
[16] Wikipedia. 2012. Flesch–Kincaid readability test. [ONLINE] Available at: http://en.wikipedia.org/wiki/Flesch%E2%80%93Kincaid_readability_test. [Accessed 03 December 12], **(2012)**.
[17] N. Abbas, "Properties of 'Good' Java Examples", Umea's 13th Student Conference in Computer Science, **(2009)**, pp. 1.
[18] C. Yung, "Simplified readability metrics", Information Systems Working Papers Series, **(1997)**.
[19] M. Halstead, "Guest Editorial on Software Science", IEEE Trans. on Sofhoare Engineering, **(1979)**, pp. 70-75.
[20] G. Chen, "A Refined Decompiler to Generate C Code with High Readability", ACM, Proceedings of the 17th Working Conference on Reverse Engineering, **(2010)**.
[21] D. Cowan, "Enhancing Code for Readability and Comprehension Using SGML", IEEE Trans. NeuralSystems and Rehabilitation Engineering, International Conference on Software Maintenance, **(1994)**, pp. 19-23.
[22] R. Baecker and A. Marcus, "Human Factors and Qpography for More Readable Programs", Addison-Wesley, Reading MA, ISBN 0-201-10745-7, **(1990)**.

[23] X. Wang, L. Pollock and K. Vijay-Shanker, “Automatic segmentation of method code into meaningful blocks to improve readability”, Reverse Engineering (WCRE), 2011 18th Working Conference, pp. 35-44, **(2011)**.
[24] P. Relf, “Tool Assisted Identifier Naming for Improved Software Readability: An Empirical Study”, IEEE Trans. Empirical Software Engineering. International Symposium, **(2005)**, pp. 112-122.
[25] S. Butler, M. Wermelinger, Y. Yu and H. Sharp, “Relating identifier naming flaws and code quality: an empirical study”, IEEE Trans. Proc. of the Working Conf. on Reverse Engineering, **(2009)**.
[26] S. Butler, M. Wermelinger, Y. Yu and H. Sharp, “Exploring the influence of identifier names on code quality: an empirical study”, Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, **(2010)**, pp. 156-165.
[27] P. Sivaprakasam and V. Sangeetha, “An accurate model of software code readability” International Journal of Engineering, vol. 1, no. 6, **(2012)**.
[28] vogella. 2009. Dijkstra's shortest path algorithm in Java. [ONLINE] Available at: http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html. [Accessed 06 December 12], **(2012)**.
[29] heimetli. 2008. Backtracking to solve a sudoku puzzle. [ONLINE] Available at: http://www.heimetli.ch/ffh/simplifiedsudoku.html. [Accessed 06 December 12], **(2012)**.
[30] freeJavaguide. 2008. date and time utility program . [ONLINE] Available at: http://www.freeJavaguide.com/Javasource4.htm. [Accessed 06 December 12], **(2012)**.
[31] R. Martin, “Clean Code, A handbook of Agile Software Craftsmanship”, Prentice Hall, **(2009)**.

Authors

Yahya Tashtoush, is an assistant professor in the computer science department at JUST University. He got his phd in 2006 from University Of Alabama In Huntsville, B.sc and MS from JUST University. His main research interests are in software engineering in general and software metrics in particular.

Zeinab Odat, is a recent master graduate from Jordan University of Science and Technology, Computer Science Department, Her main interests are in software metrics.

Izzat Alsmadi, is an associate professor in the department of computer information systems at Yarmouk University in Jordan. He obtained his Ph.D degree in software engineering from NDSU (USA). His second master in software engineering from NDSU (USA) and his first master in CIS from University of Phoenix (USA). He had B.sc degree in telecommunication engineering from Mutah university in Jordan. He has several published books, journals and conference articles largely in software engineering and information retrieval fields.

Maryan Yatim, is a recent master graduate from Jordan University of Science and Technology, Computer Science Department, Her main interests are in software metrics.