International Journal of Internet, Broadcasting and Communication Vol.16 No.3 184-191 (2024)
http://dx.doi.org/10.7236/IJIBC.2024.16.3.184

IJIBC 24-3-18

Metrics for Code Quality Check in SEED_mode.c

Jin-Kuen Hong*

Professor, Div. of Advanced IT, Baekseok University, Korea
jkhong@bu.ac.kr

Abstract

The focus of this paper is secure code development and maintenance. When it comes to safe code, it is most important to consider code readability and maintainability. This is because complex code has a code smell, that is, a structural problem that complicates code understanding and modification. In this paper, the goal is to improve code quality by detecting and removing smells existing in code. We target the encryption and decryption code SEED.c and evaluate the quality level of the code using several metrics such as lines of code (LOC), number of methods (NOM), number of attributes (NOA), cyclo, and maximum nesting level. We improved the quality of SEED.c through systematic detection and refactoring of code smells. Studies have shown that refactoring processes such as splitting long methods, modularizing large classes, reducing redundant code, and simplifying long parameter lists improve code quality. Through this study, we found that encryption code requires refactoring measures to maintain code security.

Keywords: *Code Smell, Quality, Refactor, Readability, Maintainability*

1. Introduction

In code development, readability and maintainability of code are more important than anything else. The code structure is complex, and code smells make it less readable and difficult to maintain. Code smells make code difficult to understand or modify, and appear as patterns of code with structural problems. Refactoring techniques are used to detect and remove code smells.

Metrics used for code refactoring or evaluating code quality include LOC, NOM, NOA, cyclomatic complexity (Cyclo), and maximum nesting level. These evaluation items play an important role in evaluating how complex the code is, whether it is easy to maintain, and whether it can be expanded.

In this paper, we focused on code smells and studied SEED.c, which is used for encryption and decryption. Cryptographic algorithm developers consider the safety of the code, but may relatively miss the smell nature of the code. Because discussion on these aspects is necessary, this paper examined specific encryption algorithms with the goal of increasing understanding of code smells, detecting code smells, and improving codes. Securing code quality is very important because even minor errors in security code can cause serious problems. Therefore, in this paper, we focused on indicators and evaluations to improve the code reliability and efficiency of SEED.c.

Metrics for detecting code smells are used to measure specific characteristics of code and can provide insight

Manuscript Received: June. 4, 2024 / Revised: June. 12, 2024 / Accepted: June. 17, 2024
Corresponding Author: Jin-Keun Hong
Tel:+82-41-550-2445, Fax: +82-41-550-9027
Professor, Div. of Advanced IT, Baekseok University, Korea

into code quality. In the case of SEED.c, there may be a possibility that the performance of code understanding or maintenance may be reduced due to code smells. To solve these problems, it is necessary to detect code smells and take appropriate refactoring measures. In this paper, we attempt to identify structural problems in the code and effectively improve smell characteristics through evaluation.

Chapter 2 reviews the literature and considers previous research on code smells and evaluation indicators. Chapter 3 examines the code smells that exist in SEED.c and the refactoring approach applied to solve these problems. Chapter 4 introduces metrics, focuses on the detection and analysis of code smells in SEED_mode.c, and introduces related content through a case study. Chapter 5 concludes. In this paper, we are interested in how to detect code smells contained in complex and security-sensitive code, how to improve code quality, and what metrics to use for this purpose.

2. Related Research

Lacerda et al. argue that code smells and refactoring are closely related to code quality attributes such as code understandability, maintainability, testability, complexity, functionality, and reusability. Researchers aim to identify the impact of refactoring on these quality attributes by comparing them to code smells [1]. Kokol et al. investigate the characteristics of code smells and provide insight into their nature and meaning, and it is argued that codes that identify “good smells” are worth studying [2].

Jibory addresses considerations for the detection of code smells, emphasizing that they do not eliminate smells in all code, and that tools often use combined metrics, object-oriented metrics, and metrics related to smells. Jibory classifies code smells as blotters, object-oriented abusers, mutators, distributables, and couplers [3].

Szoke et al. emphasize continuous refactoring to maintain software quality and use FaultBuster, a tool that supports automatic refactoring. FaultBuster detects problematic code parts through static analysis, uses 40 automatic algorithms to fix selected code smells, and integrates with IDEs such as Eclipse, NetBeans, and IntelliJ [4]. Verdecchia et al. hypothesize that the smell of refactored code has a positive effect on code maintainability. Researchers automatically detect and refactor five code smells, including feature envy, type checking, long methods, God classes, and duplicate code, in open source ORM-based Java applications [5].

Imran et al. explore the relationship between metrics and energy efficiency of code and claim that power consumption is significantly improved after refactoring. Researchers emphasize that after refactoring feature envy and long method smell, efficiency was improved by 49% [6]. Ournani et al. study the impact of code refactoring on energy consumption by studying seven open source software projects developed over five years. Experimental results show that code refactoring can significantly reduce energy consumption, with some projects showing a 10% reduction [7].

Refactoring is necessary to resolve code smells [8-9]. Nasrabadi et al. emphasize that the accuracy of code smell detection tools depends on the data set used [10]. A study of 45 datasets found that the suitability of a dataset for smell detection depends on its size, severity level, project type, number of each smell type, total number of smells, ratio of smelly and smell-free samples, and It was found to be largely dependent on the same function. Nucci et al. describe smells in code as a symptom of poor design and implementation choices [11]. Researchers explain that the results of many studies are subjective and tool dependent. Machine learning is applied to detect smells in code, where the learner identifies code elements with smells and code elements without smells. Despite building a dataset with 10 types of smell instances, Nucci et al. acknowledge the limitations of their research. There has been research on cyber security attack event detection in ESM using

big data and deep learning [12]. This research presents the results of a learning study in intrusion detection design, and code accuracy should also be considered in the case of the code used. Now in Chapter 3, we look at code smells and refactoring issues within the SEED.c codebase.

3. Code Smell and Refactoring of SEED.c Code

As shown in Figure 1, the metrics used to measure code smells include long methods, large classes, duplicated code, and long parameter lists, among others. For long methods, the relevant metrics are lines of code, cyclomatic complexity, and number of parameters. In the case of large classes, the metrics include the number of methods, number of attributes, and lines of code. For duplicated code, metrics such as clone detection and the number of similar blocks is used. When evaluating long parameter lists, the number of parameters is considered. For feature envy, metrics like access to foreign data and coupling between objects are relevant. These parameters are essential metrics for assessing code smells.

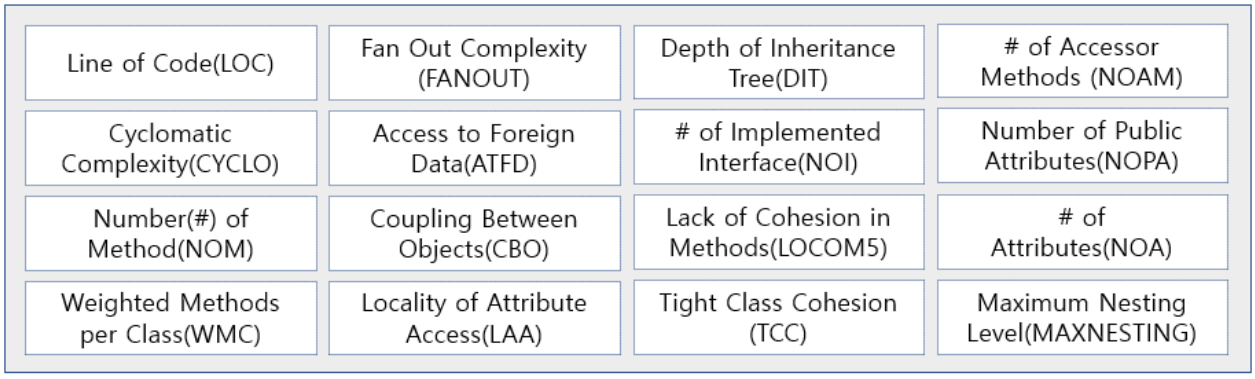

Figure 1. Metrics about code smell

As shown in Table 1, these metrics are evaluated based on mathematically quantitative values, and represented the metric indicators of code smells and the quantitative values used to evaluate these code smells.

Table 1. Mathematical expressions in metrics of code smell

Metrics	Expressions
LOC	Total number of lines in the code
NOM	Total number of methods
NOA	Total number of attributes in the class
CYCLO	E−N+2P
MAXNESTING	Maximum depth of nested blocks
FANOUT	Number of other methods / functions called by a method
ATFD	Number of accesses to attributes of foreign classes
CBO	Number of other classes to which a class is coupled
LAA	Number of accesses to own attributes / Total number of attributes accesses
NOAM	Number of accessor methods (getters and setters)
NOPA	Number of public attributes
DIT	Maximum depth of the class in the inheritance tree
NOI	Number of classes that inherit from the class

LCOM5	$\frac{1}{M}\sum_{i=1}^{M}$ (Number of attributes used by method i)
TCC	Number of connected pairs of methods / Total number of pairs of methods

Where E represents the number of edges in the control flow graph, N denotes the number of nodes, and PPP signifies the number of connected components. M refers to the number of methods in the class.

4. Evaluation Metrics and Case Study for Code Smell in the SEED_mode.c

4.1 Analysis of the Metrics Values

The CTR and CBC modes exhibit high values in LOC, NOM, NOA, and Cyclo, indicating complex structures. In contrast, the CMAC and SEED base modes feature lower values in LOC, NOA, and Cyclo, reflecting relatively simpler structures. The CCM and GCM modes have moderate values in terms of LOC and Cyclo. The ECB mode also shows moderate LOC and Cyclo values, but with a very low number of attributes.

Table 2 shows the number of code smell metrics for each mode in the SEED code. From the perspective of LOC, a higher LOC often correlates with a higher NOM, as more lines of code can accommodate a greater number of methods. Additionally, an increased LOC tends to elevate cyclomatic complexity, as more lines of code generally introduce more control flow elements.

Table 2. Numbers of metrics for code smell in each mode of SEED_mode.c code

mode	LOC	NOM	NOA	CYCLO	MAX_NESTING	FANOUT	ATFD	COB	LAA	NOAM	NOPA
ECB	483	4	1	33	2	0	0	0	0	0	1
CTR	793	16	24	71	3	3	0	0	0	0	24
CCM	314	0	4	35	3	0	0	0	0	0	4
GCM	434	0	6	38	3	0	0	0	0	0	6
CMAC	154	0	3	19	3	0	0	0	0	0	3
CBC	824	16	24	83	4	3	0	0	0	0	24

Higher LOC may correlate with an increased max nesting level, as a greater number of lines of code can result in more nested blocks. Regarding NOM, a higher number of methods often implies a higher LOC, since more methods encompass more lines of code. An elevated NOM can also lead to increased cyclomatic complexity, given that more methods typically introduce more complexity. Furthermore, a higher NOM may suggest a higher NOA, as numerous methods may necessitate more attributes.

From the perspective of NOA, a higher number of attributes is likely to correlate with an increased NOM, as more attributes tend to be utilized by a greater number of methods. Similarly, a higher NOA often results in a higher LOC, as more attributes are associated with more lines of code. In terms of Cyclo, elevated cyclomatic complexity often coincides with higher LOC, as complex code structures generally require more lines of code. A higher Cyclo can also imply an increased NOM, as complex logic typically demands more methods.

With respect to Max Nesting, an elevated max nesting level likely correlates with higher LOC, as deeply nested structures incorporate more lines of code. Higher max nesting also tends to increase Cyclo, since nested structures contribute to greater complexity. In terms of Fanout, increased fanout often correlates with higher

LOC, as more external dependencies can lead to more lines of code. Additionally, higher fanout may imply an increased NOM, as handling external dependencies often requires more methods.

Regarding NOPA, a higher number of public attributes typically suggests a higher NOA, as an increase in public attributes generally raises the overall number of attributes. A higher NOPA also likely correlates with an increased NOM, as more methods are needed to utilize public attributes.

Figure 2 shows the quantitative metrics used to identify code smells in each mode of the SEED code, depicted graphically.

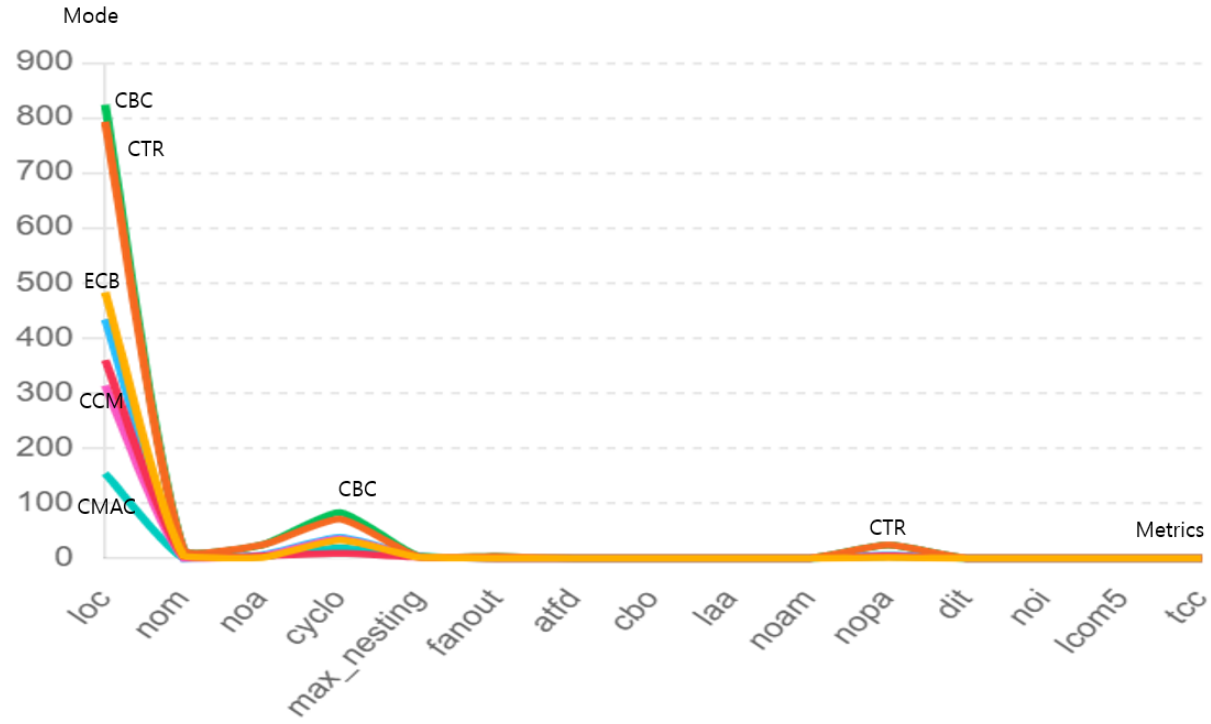

Figure 2. Numbers of each metrics of code smell vs. each mode of SEED_mode.c code

To illustrate, the CBC mode has the highest LOC, whereas the CMAC mode has the lowest LOC. Overall, the CBC mode exhibits the highest number of metrics, while the CMAC mode shows the least. The CBC mode employs a method of encryption where each data block is combined with the encrypted result of the previous block. Consequently, this mode may encompass complex sequential processing logic, along with additional error handling and initialization code, thereby increasing the complexity. In contrast, the CMAC mode processes messages in block units for message authentication, ultimately generating a single message authentication code. This process is relatively straightforward and may require fewer functions and variables.

From the correlation perspective as observed in the results of previous Table 2 and Figure 2, there is a tendency for LOC and NOM to increase together when their correlation coefficient is high. Similarly, for Cyclo and Max Nesting, a high correlation coefficient indicates that higher complexity is associated with deeper nesting levels. Additionally, for Fanout and NOM, a high correlation coefficient suggests that a greater number of external dependencies is accompanied by an increased number of methods.

4.2 Case Study of refactoring process at Code Smell of SEED_CBC.c

The following describes the case before and after the code smell in SEED_CBC mode.

input: "Hello, this is a test message."

key: 128 or 256 bits.

iv: Initialization vector.

length: The length of the input message in bytes.

The encrypt_cbc function processes the input message in blocks, performing an XOR operation between each plaintext block and the IV (or the previous ciphertext block). It then encrypts the resulting block and updates the IV with the newly encrypted block to use for the next segment of plaintext.

This method ensures that each time the same message is encrypted, the output will be different due to the unique initialization vector and the chaining process.

The following is an explanation of the code before and after refactoring.

In the case of the long method, consider case to illustrate how this function operates. Imagine we have the following inputs:

Then, the refactored (after) code extracts two new methods, xor_block and update_iv_and_output, from the original encrypt_cbc method, separating portions of its internal logic into distinct methods. The xor_block method generates a new block by performing an XOR operation between the block and IV. The update_iv_and_output method stores the encrypted block in the output array and updates the IV. The pre-refactoring (before) encrypt_cbc method is long and complex, making it difficult to understand, as it includes the logic for XOR operations, encryption, and updating the output array and IV within a single method. Post-refactoring (after), the encrypt_cbc method is shorter and clearer, with each task delegated to separate methods, thereby enhancing readability and clarity of each method's role.

For large class, in case of before, the monolithic Cipher structure consolidates all related data into one comprehensive structure. Initially straightforward, it can become cumbersome and difficult to manage as the structure's size increases. In case of after, the modular approach improves readability and maintainability. Different aspects of the encryption process are separated into distinct structures, making the codebase clearer and easier to modify. Also, in case of before, all fields are combined into one structure, obscuring the individual roles of the key, IV, and data fields. In case of after, each structure (Key, IV, Data) is dedicated to a single responsibility, adhering to the Single Responsibility Principle (SRP). This clear division of responsibilities makes each part more focused and manageable.

Table 3. Before and after refactoring of code smell in SEED_CBC.c code

Code Smell	Before	After
Long Method	**void encrypt_cbc**( ) { block[j] = input[i+j] ^ iv[j]; } **encrypt_block**(block, key);	**void xor_block**( ) { block[j] = input[offset+j] ^ iv[j]; } **void update_iv_and_output( )** {

	output[i + j] = block[j]; iv[j] = block[j];	**output**[offset+j] = block[j]; iv[j] = block[j]; } **void encrypt_cbc( )** { xor_block( ); encrypt_block( );update_iv_and_output( ); }
Large Class	**struct Cipher** { key; iv; input; output; length;}	**struct Key** { }; **struct IV** { }; **struct Data**{ }; **struct Cipher** { key; iv; data; };
Duplicated code	**void encrypt_block**(block, key) { } **void decrypt_block( )** { }	**void process_block**( ) { process(block, key); } **void encrypt_block**( ) { process_block( ); } **void decrypt_block**( ) { process_block( ); }
Long Parameter List	**void process_data**( ) { }	**struct ProcessParams** { }; **void process_data**(ProcessParams) { }
Feature Envy	**void process_and_log**( ) { encrypt_block(block, key); log_encryption(block); }	**void process_block**( ) {encrypt_block( );} **void log_block**( ) log_encryption( ); } **void process_and_log( )** {process_block( ); log_block( ); }

Cipher is initialized with a key, IV, and input data. The encrypt_cbc function then processes the input data, and the result is stored in the output field of Cipher. This structure helps organize the various elements required for the encryption process, making the code easier to manage and understand.

In the case of duplicated code, the before scenario involves a Cipher structure with numerous fields, making it a large class. Conversely, the after scenario breaks down the Cipher structure into three smaller structures: Key, IV, and Data, each with more clearly defined responsibilities.

Regarding the long parameter list, the before scenario has the process_data function with too many parameters, complicating the function call. The after scenario introduces the ProcessParams structure, consolidating the parameters into a single object, thus simplifying the process_data function.

In the feature envy context, the before scenario sees the process_and_log function handling two tasks simultaneously, creating high dependency on the encrypt_block function. The after scenario resolves this by splitting the functionality into process_block and log_block functions, allowing each to operate independently and thereby mitigating the feature envy issue.

5. Conclusion

This paper emphasizes that detecting and removing code smells is very important in maintaining the quality and security of security codes such as SEED.c. We showed that code structure can be improved by detecting code smells in SEED.c code and reducing code complexity through refactoring. For example, we focused on making the code more understandable and maintainable by modifying long methods into smaller, more maintainable functions, modularizing large classes, and consolidating long parameter lists into parameter objects. Additionally, feature envy was resolved by isolating problematic modules and reducing dependencies, improving code clarity and modularity. Future research will focus on researching automated refactoring tools and techniques that can streamline the code smell detection and removal process, making it easier to maintain and increasing reliability.

Acknowledgement

This paper is sponsored of project funding in 2024 at Baekseok University

Reference

[1] G. Lacerda, F. Petrilo, M. Pimenta, Y. G. Gueheneuc, "Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations," *The Journal of Systems and Software*, pp. 1-44. *arxiv.org/abs/2004.10777*. April 24, 2020. DOI: https://doi.org/10.1016/j.jss.2020.110610

[2] P. Kokol, M. Zorman, B. Zlahtic, G. Zlahtic, "Code Smells," Feb. 2018. *https://arxiv.org/pdf/1802.06063*.

[3] F. K. AL- Jibory, "Code Smells in software: Review," *In. J. Nonlinear Anal. App*l., Vol. 14, No. 1, pp. 1339-1345, 2023. *https://ijnaa.semnan.ac.ir/article_7029_3a5fb78be2c0a7d13bbad81f7c849b41.pdf*. DOI: http://doi.org/10.22075/IJNAA.2022.7029

[4] G. Szoke, C. Nagy, L. J. Fulop, R. Ferenc, and T. Gyimothy, "FaulBuster: An Automatic Code Smell Refactoring Toolset," In *Proc. IEEE SCAM*, pp. 253-258, 2015. *https://www.inf.szte.hu/~ncsaba/research/pdfs/2015/Szoke-SCAM2015-preprint.pdf*

[5] R. Verdecchia, R. A. Saez, G. Procaccianti, P. Lago, "Empirical Evaluation of the Energy Impact of Refactoring Code Smells," In *Proc. 5th International Conference on ICT4S*, Feb. 2018. DOI: http://doi.org/10.29007/dz83

[6] A. Imran, T. Kosar, J. Zola, F. Bulut, "Predicting the Impact of Batch Refactoring Code Smells on Application Resource Consumption," *In Proc. ACM conference'17*, July 2017. *https://arxiv.org/pdf/2306.15763*. DOI: https://doi.org/10.1145/nnnnnnn.nnnnnnn

[7] Z. Ournani, R. Rouvoy, P. Rust, J. Penhoat, "Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption," *HAL Open Access Archie*, Apr. 19, 2021. *https://hal.science/hal-03202437/document*

[8] M. Fowler and K. Bent, *Refactoring: Improving the Design of Existing Code*, 2018, 2nd ed. Addison-Wesley.

[9] 6 Bad Smells in Code - variant.ch *http://www.variant.ch/papers/IntroductionRefactoring/doc/refactoring/node7.html*

[10] M. Z.- Nasrabadi, S.Parsa, E. Esmaili, F. Palomba, "A Systematic Literature Review on the Code Smells Datasets and Validation Mechanisms," *ACM Computing Surveys*, Vol. 55, Issue 13S, No. 298, pp. 1-48, July, 2023. DOI: https://doi.org/10.1145/3596908

[11] D. di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. de Lucia, "Detecting code smells using machine learning techniques: Are we there yet?," *In Proc. IEEE 25th SANER*, pp. 612-621. March 18-23, 2018. DOI: http://doi.org/10.1109/SANER.2018.8330266

[12] H.-M. Lee, S.-J. Lee, "A Study on Security Event Detection in ESM Using Big Data and Deep Learning," *The Journal of Internet, Broadcasting and Communication*, Vol. 13, No.3, pp. 42-49, Aug., 2021. DOI: http://dx.doi.org/10.7236/IJIBC.2021.13.42