SSRG International Journal of Computer Science and Engineering
ISSN: 2348 – 8387 / https://doi.org/10.14445/23488387/IJCSE-V9I4P102

Volume 9 Issue 4, 7-13, April 2022

Original Article

Source Code Readability, Clean Code, and Best Practices: A Case Study

Lucas de Lima Silva[1], Giovani Fonseca Ravagnani Disperati[2], Antonio Angelo de Souza Tartaglia[3]

[1,2,3]*Federal Institute of Education, Science, and Technology of São Paulo, São Paulo, Brazil.*

Received: 24 February 2022 Revised: 05 April 2022 Accepted: 10 April 2022 Published: 29 April 2022

Abstract *- Maintenance of a software product during its operating lifecycle is usually the most expensive part of a project, as this phase extends indefinitely. A software project that originally met a certain set of requirements will invariably change over time since the requirements themselves tend to change. Thus, one of the main challenges of Software Engineering, particularly considering the coding activity, is establishing practices that allow greater readability of source codes to keep codebases under control. Clean code, source code refactoring, automated testing, and the application of best practices - such as design patterns - are considered starting points for this. In this article, we sought to carry out a case study based on the code kata known as Gilded Rose. A questionnaire was applied to compare programmers' understanding of a code without best coding practices to understanding a refactored code using best practices. We conclude that student or intern-level programmers most beneficiated from such practices as a more readable code imposes less of a barrier to their understanding of the code itself and the functional requirements implemented by such code.*

Keywords *- Clean Code, Code implementation, Design Patterns, Software Engineering.*

1. Introduction

Techniques and tools for coding quality software projects are one of the main study points in Software Engineering. Quality software can be defined as one that attains the functional and non-functional requirements, follows a consistent style, is easy to understand, is well documented, and can be tested [4], [5], [6], [7]. Such characteristics, therefore, make quality software easier to maintain throughout its life cycle, which is usually the longest phase in any software project [1], [3]. Maintainability is essential since a software project usually has a closed set of functional and non-functional requirements that must be reached in its initial conception. Still, over time, such requirements tend to change due to changes in the business rules, technologies, and external regulations, among other factors, which causes the codebase to be modified to meet these new requirements. A software project that is intrinsically hard to modify will present challenges in the long run. For this reason, it becomes evident that software engineers must focus on maintainability.

One of the main factors contributing to the maintainability of a software project throughout its life cycle – that is, one of the main factors for obtaining quality software – is readable code. Software projects have two distinct value types: behavioral and structural values. The behavioral value of software is what it delivers in the present, that is, the fulfillment of current business requirements. The structural value, in turn, presents itself concerning the possibility of software to continue meeting the requirements throughout its life cycle. The structural value of a software project concerns its architecture, whose maintenance throughout the life cycle is also correlated with the readability and ease of understanding of the codebase [5]. Most of the efforts employed in software maintenance are related to new features and fault correction. The consequence is a decline in software quality as the code quality and structure degrade as modifications occur. At the beginning of its life cycle, software that needs a certain number of people to maintain will require more people over time due to the degradation of the quality of its source code. A hard-to-maintain code base will be costlier.

It is a serious problem in the software industry because the high complexity sometimes makes the project unfeasible. In this sense, low quality is one of the main reasons for extrapolating the schedule of software projects, and the lack of quality is responsible for the cancellation of half of the projects. In the same way that computer systems can generate competitive advantages, they can also cause an organisation's ruin. With those, new concerns arise concerning the quality of the code developed.

Thus, techniques and methods emerge proposing code production and maintenance practices to maintain code quality throughout its life cycle. Among these, clean code, refactoring, and best practices stand out in particular. Clean code can be defined as methods and guides for writing source code focusing on its readability and understanding [1]. Refactoring is a process that consists of changing the source code without changing the software's behavior, which can be used to achieve clean code [9], [10]. Refactoring can also be understood as a change made to the software structure to make it easier to understand and cheaper to modify without changing its observable behavior. Good practices, in turn, refer to a more general practical-theoretical set of practices that involves the construction of automated tests for the application, the use of principles known as SOLID, and the use of design patterns, among others [2], [4]. Those methods have been extensively studied in the literature [11-25].

In this context, a case study is proposed to evaluate the understanding of a source code without using such practices versus a refactored code using techniques focused on providing improved legibility and ease of understanding. A questionnaire was made to conduct this case study. Volunteers were asked questions concerning their understanding of a code kata known as Gilded Rose before and after its refactoring process.

The case study was focused mainly on undergraduate students of the systems analysis and development course of São Paulo's Federal Institute of Science and Technology Campus of Guarulhos, São Paulo. Professional analysts were also respondents to the questionnaire. They provided a baseline of comparison for the following research question: will an easier-to-understand code be more beneficial to students and junior analysts?

2. Materials and Methods

The case study of this software included a refactor of a pre-existing codebase known as Gilded Rose, which is typically used as a challenge for programmers to practice refactoring. The codebase is well-known to be challenging and purposefully hard to understand, with the business rules appearing under nested if-else clauses and cryptical names. It was also the code in which refactoring based on the techniques known as good practices was implemented. After the refactoring process, a questionnaire with questions focused on impressions about the impacts that the refactoring had on the code was made [8]. The applied questionnaire consists of 10 questions with multiple choice answers and an optional comment at the end, where the participants can share their perceptions about the codebase before and after the refactor. The step was to identify the participant's experience level to compare results. After that, a brief introduction about the code was made to present the following questionnaire.

The first question presented the code in Fig. 1 (a). and Fig 1 (b), shown in Appendix A, which introduces the GildedRose class. It is important to notice that in the codebase, before refactoring, every business rule of the Gilded Rose code kata is implemented in this GildedRose class.

Based on the above code, it was asked, "Observing the code, how hard was it for you to understand the functional requirements?". The objective was to identify how difficult it was to understand what the unaltered code did, using a 4-point Likert scale, (1) very difficult, (2) difficult, (3) easy, (4) very easy. That was a relevant question since only a brief introduction about the code was made, and no further clarifications were provided. That may be a common occurrence in a real-world scenario of legacy code. Thus some degree of comprehension of the functional requirements obtained after reading the code base is an important aspect of a maintainable software project.

The second question, "How long did it take you to understand the code above?" asked the participant how long it took to understand the code presented in Fig. 1, with 4 alternatives, (1) up to 2 minutes, (2) up to 10 minutes, (3) up to 20 minutes, (4) more than 30 minutes.

To ensure that the code in Fig. 1 (a). and Fig. 1 (b) was interpreted, the third question was formulated as "Look again at the code snippet above. What will happen when the item is Sulfuras, Hand of Ragnaros?" with 3 alternatives, being (1) "The item does not have its sale date or quality changed," (2) "The item increases its quality from zero," (3) "Item increases its quality when the quality is lower than 50". The correct answer is, in this case, alternative (1).

After this question, we introduce the refactored code in the questionnaire. Fig. 2, shown in Appendix A, shows a refactored code snippet.

Fig. 2 presents the result of the refactor, which implemented the Simple Factory pattern, creating objects from the classes of items implemented with the Strategy pattern. This pattern was chosen because it facilitates the separation of the individual rules of each item in its specific class, thus allowing each class to have only one responsibility, that of dealing with the individual rules of each item, as defended by the principle of single responsibility.

After presenting the refactored code, we reintroduced the code in Fig. 1 and highlighted each if statement with an item's name. We then asked the fourth question, "Do you consider the highlighted code snippets easier to understand after the refactor?". Then, the fifth question was asked concerning the refactored code "How long did it take you to understand the code above?".

Then, the non-refactored code was reintroduced, and a direct comparison against the refactored one was made. Fig 3., shown in Appendix A, shows the code snippet reintroduced in question five.

Based on the code snippet in Fig. 3, the sixth question asked, “Looking at the code snippet below, what do you understand?” with the following alternatives: (1) “The code is adding the item "Backstage passes to a TAFKAL80ETC concert" in a list," (2) “The code is removing the item "Sulfuras, Hand of Ragnaros," (3) “The code is updating the "Aged Brie" item." The correct answer is, in this case, alternative (3).

As the seventh question, it was asked again about the snippet in Fig. 3, “How long did it take you to understand the code above?” with 4 alternatives, (1) up to 2 minutes, (2) up to 10 minutes, (3) up to 20 minutes, (4) more than 30 minutes. After that, for question 8, the refactored code shown in Fig. 4 was introduced.

It was then asked, as the eighth question, what the code snippet was doing, “What is the refactored code of the AgedBrieStrategy class doing?” and the following alternatives were presented: (1) “Increasing Quality if it is less than the maximum quality," (2) “Reducing Sellin if the quality is less than the maximum quality," (3) “Increasing Sellin if the quality is higher than the maximum quality." The correct answer is, in this case, alternative (1).

The ninth question concerned the code snippet in Fig. 4, shown in Appendix A, and asked, “How long did it take you to understand the code above?" with 4 alternatives, (1) up to 2 minutes, (2) up to 10 minutes, (3) up to 20 minutes, (4) more than 30 minutes.

Finally, the tenth question was an open field where the participants could express their thoughts about the presented codes: “Based on the pre-refactoring and post-refactoring code, speak freely about your understanding of both versions.”

3. Results

The questionnaire was shared with professionals who work as software engineers and students in areas related to information technology. In total, twenty-six participants answered the questionnaire.

Of those twenty-six, seventeen, or 65.4%, were students or intern level. Five, or 19.2%, were junior developers, one, or 3.8%, was a developer, and three, or 11.5%, were senior developers.

For the first question, “Observing the code, how hard was it for you to understand the functional requirements?” nineteen, or 73.1%, said it was “hard." The three senior developers said the code was easy or easy to understand, and only two student or intern-level developers said it was easy to understand. The vast majority of students or intern-level developers, fifteen out of seventeen, said it was either hard or hard to understand to code presented in Fig. 1.

For the second question, nineteen out of the twenty-six participants took 10 minutes to understand the code.

The third question, which verified if the participant had understood the code presented in Fig. 1, had fourteen correct answers. Eight out of the seventeen students or intern-level developers, or 47.06% of this group, had an incorrect answer.

For the fourth question, every participant agreed that the refactored code presented in Fig. 2 was easier to understand. In the fifth question, eighteen out of the twenty-six said it took them until two minutes to understand the code.

For the sixth question, seventeen out of the twenty-six had a correct answer, including twelve out of the seventeen student or intern-level developers.

For the seventh question, sixteen out of the twenty-six participants took ten minutes to understand the code.

In the eighth question, however, twenty-four out of the twenty-six participants had a correct answer after presenting the refactored code. Twenty-one of them took two minutes to understand the code presented in Fig. 4, as answered in question nine.

4. Conclusion

The answers support the theoretical aspect of the case study: an easier-to-read code base is easier to understand. It is supported by the theory and the established practices in the Software Engineering field.

The questions analyzed in Figure 4 demonstrated that even without the refactoring process, the simple fact that the code was displayed focused on a code snippet improved the results. It may indicate that theories, heuristics, and techniques, which defend the simplicity and objectivity of the code, contribute to readability.

The questions that analyzed the refactored code brought great results, as it was possible to reduce the time to understand it without impairing the understanding of the code. Based on these results, it is evident that the use of clean code heuristics and best practices increased the readability and quality of the code.

It also points towards the importance of readability n the code base to lower the required skill level of developers involved in the project. The results obtained show that it was evident that the students or intern-level developers were the most beneficiated by the refactoring in the codebase. This group had the most difficulties understanding the non-refactored code base, as it could be verified, especially in question three. The ratio of students or intern-level developers with incorrect answers in question three was much higher than that of the question. In the tenth and final question, where participants could speak freely about the code, most praised the refactored code and talked about how much easier it was to understand.

It is concluded, thus, that the importance of a readable code base is not only due to maintainability in its lifecycle but also as a means of lowering the skill level required of a working developer in a software project. However, this raises another question: would a group of student or intern-level developers be able to maintain a readable codebase using the refactoring techniques presented not only in this case study but also throughout the literature? Further studies in this sense might be made to explore this question.

Appendix A

In this appendix, the source code used for the questionnaire is presented. Fig. 1 (a) and Fig. 1 (b) present the non-refactored version of the Gilded Rose Class. Fig 2. presents the refactored code implementing the strategy pattern. Fig 3. presents non-refactored if-else statements. Fig 4. presents the AgedBrieUpdateStrategy class. Each figure is presented sequentially.

```
using System.Collections.Generic;

namespace csharp
{
    public class GildedRose
    {
        IList<Item> Items;
        public GildedRose(IList<Item> Items)
        {
            this.Items = Items;
        }

        public void UpdateQuality()
        {
            for (var i = 0; i < Items.Count; i++)
            {
                if (Items[i].Name != "Aged Brie" && Items[i].Name != "Backstage passes to a TAFKAL80ETC concert")
                {
                    if (Items[i].Quality > 0)
                    {
                        if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
                        {
                            Items[i].Quality = Items[i].Quality - 1;
                        }
                    }
                }
                else
                {
                    if (Items[i].Quality < 50)
                    {
                        Items[i].Quality = Items[i].Quality + 1;

                        if (Items[i].Name == "Backstage passes to a TAFKAL80ETC concert")
                        {
                            if (Items[i].SellIn < 11)
                            {
                                if (Items[i].Quality < 50)
                                {
                                    Items[i].Quality = Items[i].Quality + 1;
                                }
                            }

                            if (Items[i].SellIn < 6)
                            {
                                if (Items[i].Quality < 50)
                                {
                                    Items[i].Quality = Items[i].Quality + 1;
                                }
                            }
                        }
                    }
                }

                if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
                {
                    Items[i].SellIn = Items[i].SellIn - 1;
                }

```

Fig. 1 (a) The GildedRose class before refactoring

```
            if (Items[i].SellIn < 0)
            {
                if (Items[i].Name != "Aged Brie")
                {
                    if (Items[i].Name != "Backstage passes to a TAFKAL80ETC concert")
                    {
                        if (Items[i].Quality > 0)
                        {
                            if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
                            {
                                Items[i].Quality = Items[i].Quality - 1;
                            }
                        }
                    }
                    else
                    {
                        Items[i].Quality = Items[i].Quality - Items[i].Quality;
                    }
                }
                else
                {
                    if (Items[i].Quality < 50)
                    {
                        Items[i].Quality = Items[i].Quality + 1;
                    }
                }
            }
        }
    }
}
```

Fig. 1 (b) The GildedRose class before refactoring (continuation of the source code)

```
public class UpdateItemStrategyFactory : IupdateItemStrategyFactory
{
    public IStockItemUpdateStrategy  Create(Item item)
    {
        if (item == null)
        {
            throw new ArgumentNullException(nameof(item));
        }

        switch (item.name)
        {
            case "Aged Bried":
                return new  AgedBrieUpdateStrategy();
            case "Backstage Passes to a TAFKAL80ETC concert":
                return new  BackstagePassesUpdateStraty();
            case "Sulfuras, Hand of Ragnaros":
                return new  LegendaryItemsupdate();
            case "Conjured Mana Cake":
                return new ConjuredUpdateStrategy();
            default:
                return new StandardItemsUpdateStrategy();
        }
    }
}
```

Fig. 2 Refactored code implementing the Strategy pattern.

```
if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
{
    Items[i].SellIn = Items[i].SellIn - 1;
}

if (Items[i].SellIn < 0)
{
    if (Items[i].Name != "Aged Brie")
    {
        if (Items[i].Name != "Backstage passes to a TAFKAL80ETC concert")
        {
            if (Items[i].Quality > 0)
            {
                if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
                {
                    Items[i].Quality = Items[i].Quality - 1;
                }
            }
        }
        else
        {
            Items[i].Quality = Items[i].Quality - Items[i].Quality;
        }
    }
    else
    {
        if (Items[i].Quality < 50)
        {
            Items[i].Quality = Items[i].Quality + 1;
        }
    }
}
```

Fig. 3 Non-refactored nested if-else statements

```
public class AgedBrieUpdateStrategy : IStockItemUpdateStrategy
{
    public void UpdateItem(Item item)
    {
        item.SellIn--;

        const int maxQuality = 50;
        if (item.Quality < maxQuality)
            item.Quality++;
    }
}
```

Fig. 4 The Aged Brie Update Strategy class

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Boston, Usa: Addison-Wesley, 2019.

[2] S. N. Varela, "Source Code Metrics: A Systematic Mapping Study," *The Journal of Systems and Software*. [S. L.] , pp. 128-197, 2017.

[3] E. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st Edition Boston, USA: Addison-Wesley, 1994.

[4] R. C. Martin, *Clean Code a Handbook of Agile Software Craftsmanship*,Massachusetts, USA: Prentice Hall, 2009.

[5] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, London, England: Pearson, 2017.

[6] K. Beck, *Test Driven Development*: by Example, Bonton, USA: Addison-Wesley, 2002.

[7] H. Q. Jaleel, "Testing Web Applications," *SSRG International Journal of Computer Science and Engineering (SSRG-IJCSE)*, vol. 6, no. 12, pp. 1-9, 2019. Crossref, https://doi.org/10.14445/23488387/IJCSE-V6I12P101

[8] M. K. Yogi, G. Kavya, "Novel Empirical Methods for Advance Software Engineering Research," *SSRG International Journal of Computer Science and Engineering (SSRG-IJCSE)*, vol. 5, no.7. pp. 1-7, 2018. Crossref, https://doi.org/10.14445/23488387/IJCSE-V5I7P103

[9] B. Latte, S. Henning, M. Wojcieszak, "Clean Code: on the Use of Practices and Tools to Produce Maintainable Code for Long-Living Software," *In EMLS 2019: 6th Collaborative Workshop on Evolution and Maintenance of Long-Living Systems*, pp. 96-99, 2019.

[10] H. G. Koller, *Effects of Clean Code on Understandability: An Experiment and Analysis*, University of Oslo, pp. 1-100, 2016.

[11] J. G Ganssle, "Tools for Clean Code," *Embedded Systems Programming*, vol. 14, no. 4, pp. 177-180, 2021.

[12] S. W. Ambler, J. Jaros, J. Highsmith, K. Guntheroth, E. A. Weiss, S. Pothier, "One Sure Thing is Good Clean Code," *Communications of the ACM*, vol. 44, no. 12, pp. 11-13, 2001.

[13] J. V. Petersen, "Ten Reasons Why Unit Testing Matters," *Code Magazine*, pp. 70-73, 2019.

[14] P. Rachow, S. Schröder, M. Riebisch, "Missing Clean Code Acceptance and Support in Practice-An Empirical Study," *2018 25th Australasian Software Engineering Conference* (*ASWEC*), pp. 131-140, 2018.

[15] J. Katona, "Clean and Dirty Code Comprehension by Eye-Tracking Based on Gp3 Eye Tracker," *Acta Polytechnica Hungarica*, vol.18, no. 1, pp. 79-99, 2021.

[16] J. Katona, "Examination of the Advantage of the Clean Code Technique by Analyzing Eye Movement Parameters," *Proceedings of ISER International Conference*, vol. 25, no. 26, pp. 51-54, 2022.

[17] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, P Van Gorp, "Refactoring: Current Research and Future Trends," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 3, pp. 483-499, 2003.

[18] T. Mens, T. Tour, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126-139, 2004.

[19] F. Simon, F. Steinbruckner, C. Lewerentz, "Metrics Based Refactoring," *Proceedings Fifth European Conference on Software Maintenance And Reengineering. IEEE*, pp. 30-38, 2001.

[20] M. Kim, T. Zimmermann, N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1-11, 2012.

[21] G. Lacerda, F. Petrillo, M. Pimenta, Y. G. Guéhéneuc, "Code Smells And Refactoring: A Tertiary Systematic Review of Challenges and Observations," *Journal of Systems and Software*, vol. 167, pp. 110610, 2020.

[22] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, W. Oizumi, "Refactoring Effect on Internal Quality Attributes: What Haven't They Told You Yet?" *Information and Software Technology*, vol. 126, pp. 106347, 2020.

[23] I. Karac, B. Turhan, "What Do We (Really) Know About Test-Driven Development?" *IEEE Software*, vol. 35, no. 4, pp. 81-85, 2018.

[24] A. Tosun, M. Ahmed, B. Turhan, N. Juristo, "On the Effectiveness of Unit Tests in Test-Driven Development," *Proceedings of the 2018 International Conference on Software And System Process*, pp. 113-122, 2018.

[25] M. M. Moe, "Comparative Study of Test-Driven Development (TDD), Behavior-Driven Development (BDD) And Acceptance Test-Driven Development (ATDD)," *International Journal of Trend In Scientific Research And Development*, pp. 231-234, 2019.