Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

Analysis of The Impact of Software Detailed Design on Mobile Application Performance Metrics

Amal Khairin[*], Dana Sulistyo Kusumo, Yudi Priyadi
Informatics, School of Computing, Telkom University, Bandung, Indonesia
Email: [1,*]amalkhairin@student.telkomuniversity.ac.id, [2]danakusumo@telkomuniversity.ac.id,[3]whyphi@telkomuniversity.ac.id
Correspondence Author Email: amalkhairin@student.telkomuniversity.ac.id
Submitted: **21/06/2022**; Accepted: **28/06/2022**; Published**: 30/06/2022**

Abstract−The rapid development of mobile applications in recent years has forced developers to develop their applications quickly. Application quality must be one of the top concerns of developers since poor application design will affect the quality of the application. Performance is one of the important quality attributes that determine the quality of an application. One approach that can be used to overcome performance problems is the design pattern. However, as the research progressed, other approaches were discovered such as refactoring code smells and design principles. In this study, a detailed design analysis was carried out on the source code of the mobile application by applying design patterns, refactoring the code smells, and implementing design principles to determine their impact on the application's performance. To measure the application performance, Central Processing Unit (CPU) usage, memory usage, and frame rate metrics are used. Based on the implementation design patterns, refactoring the code smells, and applying design principles, the result found that design patterns can affect application performance depending on the design pattern used. The Strategy pattern and Visitor pattern optimize memory usage by 1%, while the Bridge pattern increases memory usage by 2%. Meanwhile, the result of refactoring the code smells can optimize CPU usage by 35% and memory usage by 2%, and design principles can optimize CPU usage by 25% and application frame rates by 5 frame per second (fps).

Keywords: Performance Metrics; Design Pattern; Code smells; Design principles; Refactoring; Mobile Applications

1. INTRODUCTION

The development of mobile applications in recent years has grown rapidly [1]. According to statistics provided by Statista, the number of downloads of mobile applications worldwide from 2016 to 2020 increased by more than 50%. Therefore, developers must develop applications quickly in order to keep up with rapid technological developments [2]. However, poor design can have a negative impact on application quality [2]. One of the quality attributes of software is performance [3]. The quality of software can be measured based on performance estimates such as application speed, latency, network error, and so on [4]. The effectiveness and efficiency of an application can determine its performance and increase user satisfaction [3]. This makes performance issues very important.

There are two phases in software design, namely architectural design and detailed design [5]. Both phases are very important to do to evaluate the quality of the application. The detailed design phase will define the components and design of the system in detail. Application developers are trying to determine the best design approach to build applications that are effective and efficient in terms of performance in the midst of rapid technological developments [3]. One of the design approaches that can be used by developers to overcome performance problems is the design patterns. Design patterns can solve problems in software development using object-oriented design principles [5]. however, as this research progressed, another approach was found that can be used to solve application quality problems based on the design problems found, namely refactoring the code smells and implementing design principles. Code smells and violations of design principles can be overcome by refactoring the source code [6],[7].

Many studies have been conducted related to this research problem. Aggarwal et al [4] suggests that the quality of an application can be controlled by evaluating performance estimates based on different performance criteria. This study also adds that application performance can be estimated by utilizing different criteria such as application speed, application latency, and network error. Mahendra et al [8] have conducted research on the performance of mobile applications by comparing several metrics such as Central Processing Unit (CPU) and memory usage, response time, frame rate, and application size between Flutter, React Native, and Native Android applications. In another study, Qasim et al [3] further investigated performance in terms of energy use by applying several design patterns such as Façade, Observer, and Abstract Factory patterns. The study found that the performance of mobile applications in terms of power usage can be optimized by applying design patterns. Meanwhile, Hect, G. et al [9] identified performance metrics in their research on the effect of code smell on performance. The study describes the metrics used to measure performance, namely Frame Time, Number of Delayed times, Memory Usage, and Number of Garbage Collection Calls. Willocx, M. et al [10] also describes the metrics used to measure performance in mobile applications, namely Response Time, CPU Usage, Memory Usage, Disk Space, and Battery Usage.

Gamma et al [11] explained that the design pattern is communication between classes and objects that are adapted to solve general design problems in a particular context. There are 23 design patterns categorized based on their intent, namely Creational Patterns, Structural Patterns, and Behavioral Patterns [12]. Design problems arise when the selection of software design has a bad influence on quality attributes [13]. Some examples of design problems are Cyclic Dependency, Scattered Concern, Ambiguous Interface, and Fat Interface. Several strategies can be used by developers in identifying design problems, namely smell-based, problem-based, principle-based, element-based, quality attribute-based, and pattern-based [13]. Code smells are a symptom that is an indication of the emergence of code or design problems in the source code [6],[14]. Code smells can infect a method, a class, or the entire code and

Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

can cause bugs and also increase the risk of code maintenance which results in high maintenance costs [6]. Code smells can be divided into several types such as Bloaters, Object-Oriented (OO) Abusers, Change Preventers, Dispensables, Couplers, and Incomplete Library Classes [15]. The software design principle also is the basis for creating quality attributes. The design principle will maximize profit and reduce development costs [16]. There are several Object Oriented design principles [17],[7] , namely Single Responsibility, Open-Closed, Liskov Substitution, Dependency Inversion, Interface Segregation, Separation of Concerns, Law of Demeter, DRY (Don't Repeat Yourself), and YAGNI (You Ain't Gonna Need It).

Research [5],[6],[7],[18] discusses software architecture and software design such as design patterns, design principles, and code smells that are interconnected with each other in determining the quality of a software in terms of development, performance, and maintainability. Therefore, further research is carried out to determine the relationship between the design pattern approach, code smells, and design principles in influencing application quality, especially on mobile application performance.

This study aims to analyze the impact of the application of design patterns, design principles, and refactoring of the code smells on the change of the value of mobile application performance metrics measured using the average value of Central Processing Unit (CPU) usage, memory usage, and frame rate. This study was also conducted by comparing the results of each approach to changes in the value of mobile application performance metrics measured using several metrics such as CPU usage, memory usage, and frame rate before and after implementation.

2. RESEARCH METHODOLOGY

2.1 Research Stages

This research was conducted based on the stages shown in Figure 1.

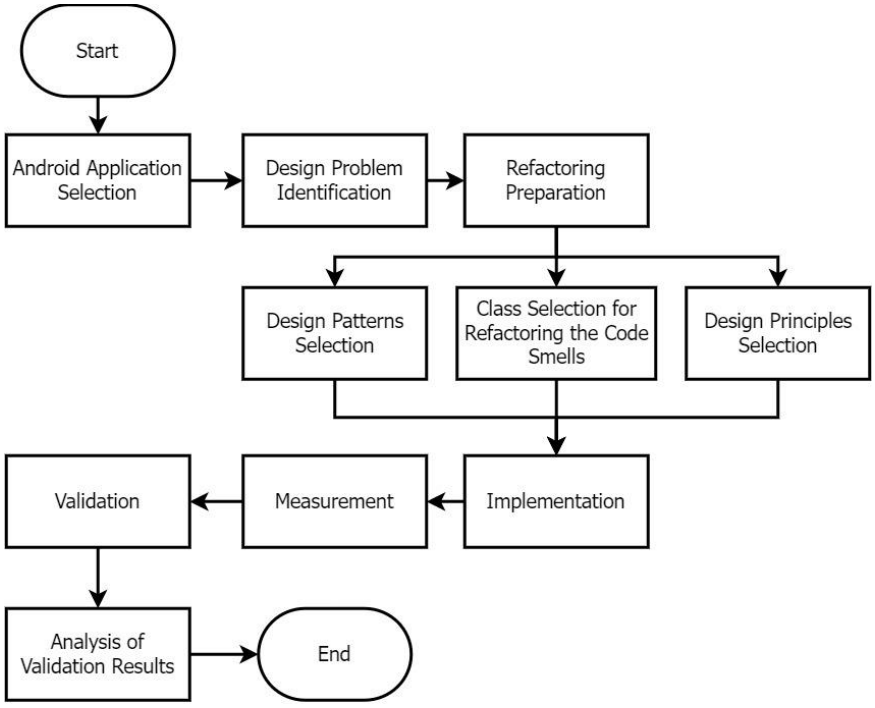

Figure 1. Research Stages

Based on the research stages in Figure 1, the first stage is to select an android application to be used as a test application. Second, identify the design problem with the source code of the application selected in the first stage. Third, make preparations before refactoring. At this stage, it is divided into three parts, namely the selection of design patterns, selection of classes for refactoring code smells, and selection of design principles. Fourth, implement. At this stage, the implementation is carried out in three scenarios, namely the implementation of a design pattern, the implementation of refactoring code smells, and the implementation of design principles to the application source code that has been selected in the first stage. Fifth, re-measure the application performance metric value after the fourth stage of implementation. The sixth is to validate the metric values before and after implementation. Finally, analyze the results of the implementation and measurements that have been carried out.

2.2 Android Application Selection

In this study, the application is selected from the Github repository which provides several open-source Android applications [19]. The selected application has a repository with a large number of Forks and Stars indicating the

Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

application is used by many developers. Based on these criteria, an android application called GSYGithub was selected [20]. The GSYGithub application is an unofficial application for the Github client on the mobile platform. The selected application is then tested to see the average percentage value of CPU usage, memory usage, and frame rate.

2.3 Design Problem Identification

The source code of the application that has been selected in section 2.2 is identified to get the design problems that exist in the application. Identification of design problems is carried out using several approaches such as Principle-based and smell-based strategies [13]. This strategy will see and analyze classes that violate design principles and have code smells. The results of identification using smell-based and principle-based strategies for application source code are shown in Tables 1 and 2.

Table 1. The results of the identification of design problems using smell-based strategy

No.	Code Smell	Design Problem	Related Class
DP-1	Object Orientation Abusers	Complex switch statements	CommonListPage
DP-2	Dispensables	Too many unnecessary comments on the code	MyPage, PhotoViewPage, LoginBloc, FlutterReduxApp, HttpErrorListener, TrendPage
DP-3	Bloaters	A long method or a function has too many lines of code	UserProfileState, HomeDrawer

Table 1 shows the results of identifying design problems using a smell-based strategy or code smells. The results also show a different type of code smell for each design problem that is obtained, as well as the related classes that have a code smell in the source code of the test application.

Table 2. The results of the identification of design problems using principle-based strategy

No.	Violated Principle	Design Problem	Related Class
DP-4	Single Responsibility	The class has some functions that it shouldn't have	CommonListPage, TrendPage
DP-5	Liskov Substitution	Some functions are too long so they can be broken down into several child classes without changing the logic of the code	UserProfileState

Table 2 shows the results of identifying design problems using a principle-based strategy or design principles. The results also show the type of principle that is violated, causing design problems by the related class.

Based on tables 1 and 2, there are five design problems based on smell-based and principle-based strategies. This design problem is used to select the appropriate design pattern to be implemented in the test application source code.

2.4 Refactoring Preparation

2.4.1 Design Patterns Selection

Gamma et al [11] describe approaches that can be taken in choosing a design pattern. In this study, the design pattern was selected based on the results of the analysis according to the design problem, intent, and connection with other design patterns. The results of the analysis can be seen in Table 4.

Table 3. The results of the selection of design patterns based on similarities with the design problem

Design Pattern	Intent	Related Design Problems
Bridge Pattern	Dividing a large class or a set of closely related classes into two separate abstractions and implementations that can be developed independently	DP-3, DP-5
Strategy pattern	Defining groups of algorithms and placing each algorithm in a separate class to	DP-1, DP-5

Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

	make those objects interchangeable
Visitor Pattern	Separating algorithms from the objects on which they operate

Based on the results of the analysis in Table 4, it is found that the design patterns that match the problems that have been obtained are the Bridge pattern, Strategy pattern, and Visitor Pattern. Figure 2 shows the structure of the Bridge pattern, Figure 3 shows the structure of the Strategy pattern, and Figure 3 shows the structure of the Visitor pattern.

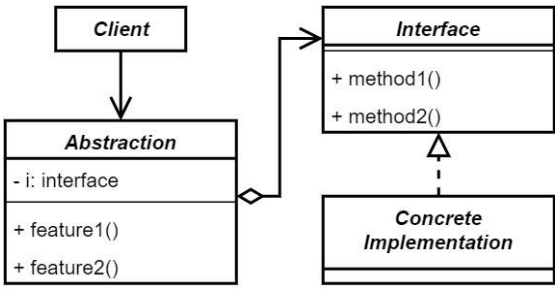

Figure 2. Bridge pattern structure

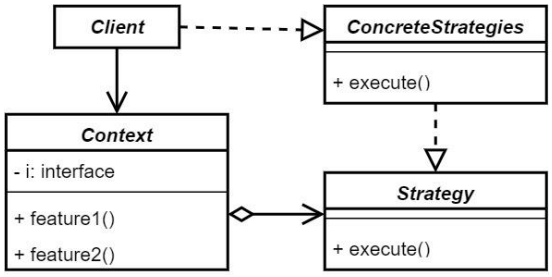

Figure 3. Strategy pattern structure

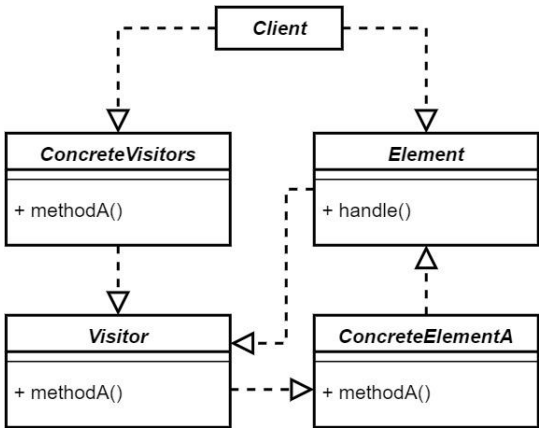

Figure 4. Visitor pattern structure

2.4.2 Class Selection for Refactoring The Code Smells

Based on the results of the identification of design problems using smell-based strategy that have been carried out, Table 1 shows the results of identifying design problems based on smell-based or code smells strategies. Based on these results, three types of code smells were found in the source code of the test application, namely Object-Orientation Abusers, Dispensables, and Bloaters. Each of the code smells that have been identified also obtained classes related to the code smell.

2.4.3 Design Principles Selection

Based on the results of the identification of design problems using principle-based strategy in Table 2, two types of design principles were found that were violated in the source code, namely Single responsibility and Liskov substitution, as well as related classes for each principle. Based on these results, the implementation is carried out by refactoring using the design principle in accordance with the principle being violated.

Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

2.5 Implementation

The predetermined design pattern will be applied to the specified test application source code. Based on the design problem, twelve classes violated the design principle and code smell. The application of the design pattern to the three classes is done by refactoring the application source code. The implementation of design patterns is done by separately applying the Bridge pattern, Strategy pattern, and Visitor pattern in each class according to the design problem in Table 3. An example class diagram of the implementation of the Bridge pattern in the *CommonListPage* class from the source code is shown in Figure 5. Meanwhile, Figure 6 shows an example class diagram of the results of the Strategy pattern implementation, and Figure 7 shows an example class diagram of the results of the Visitor pattern implementation.

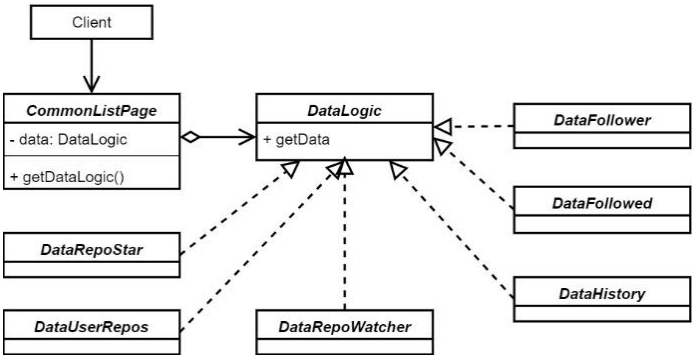

Figure 5. Class diagram of the implementation of the Bridge pattern in the *CommonListPage* class

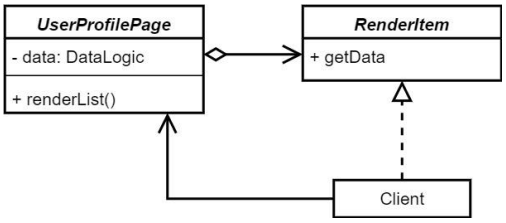

Figure 6. Class diagram of the implementation of the Bridge pattern in the *UserProfilePage* class

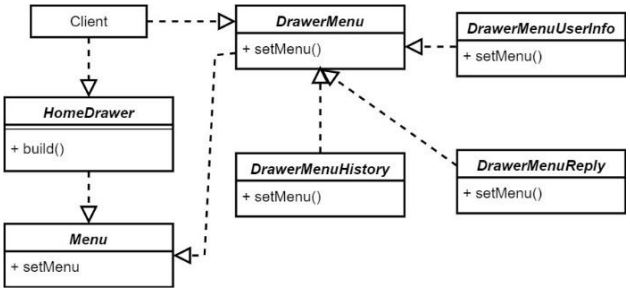

Figure 7. Class diagram of the implementation of the Bridge pattern in the *HomeDrawer* class

Refactoring was carried out on nine classes identified as having code smells based on the identification results in Table 1. These classes were then refactored based on the code smells contained in that class. An example of the results of refactoring based on Bloaters code smell can be seen in Figure 8. Methods that are indicated to have Bloaters are broken down into classes by implementing abstractions.

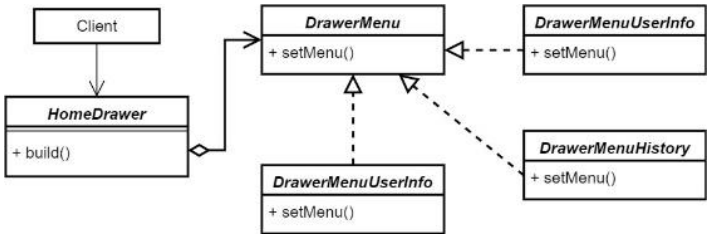

Figure 8. Class diagram of the results of refactoring the Bloaters smell code in the *HomeDrawer* class

The implementation of design principles is done by refactoring the source code based on the results of the design problem identification in Table 2. There are two types of principles that are violated in the test application source code, namely Single responsibility and Liskov substitution. Implementation is carried out on related classes

Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

based on the principle being violated. An example of a class diagram of the results of the implementation of Single responsibility can be seen in Figure 9.

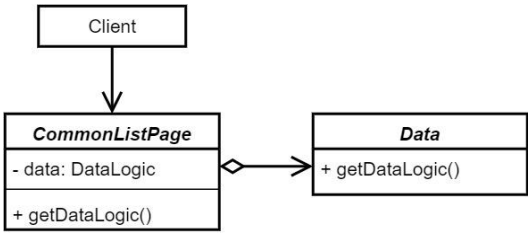

Figure 9. Class diagram of the implementation of the Single Responsibility in the CommonListPage class

2.6 Measurement

In this study, measurements were carried out twice, before and after the implementation of the design pattern. Measurements are made using performance metrics, namely CPU usage, memory usage, frame rate, and response time to the test application. The tools used to take measurements are GPU Watch to see the percentage of average CPU usage and frame rate, and ADB Shell "dumpsys meminfo" to see memory allocation by the test application. Measurements were carried out using the scenario in Figure 10. The device used to perform performance analysis has specifications as shown in Table 3.

Tabel 4. Device specifications used to perform performance analysis

Device Name	Operating System	Screen Resolution	Memory (RAM)	Internal Storage	CPU
Galaxy A30s	Android 11	720 x 1560 Pixel	4GB	64GB	Exynos 7904, 449 MHz-1768 MHz

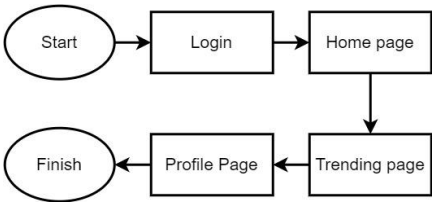

Figure 10. Scenarios used in measuring performance in test applications

2.7 Validation

Validation is done by comparing the results of measuring the performance metric values before and after implementation. The comparison results are then analyzed to see the effect of implementing design patterns, design principles, and refactoring on code smells on performance on mobile applications.

2.8 Analysis of Validation Results

After validating, the validation results are then analyzed to see the effect of the design pattern on the performance metric value after implementing the design pattern on the android application.

3. RESULT AND DISCUSSION

3.1 Measurement Results

At this stage, it explains the measurement results of each performance metric such as CPU usage, memory usage, frame rate, and response time after implementing design patterns, refactoring code smells, and implementing design principles on the source code of the test application. Table 5 shows the results of measurements made before implementation.

Table 5. Measurement of the initial value of each performance metric before implementation

CPU Usage	Memory Usage	Frame Rate (*Frame per Second*)
74%	75%	50 fps

Table 5 is the result of measurements made throughout the application before implementation. The measurement measures the initial value of each performance metric. For average CPU usage, it produces 74% which means that the CPU usage in the application is high. While the memory usage is 75% and the frame rate is 50 frame per second (fps).

Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

3.1.1 Design Patterns Results

The results of the implementation of Bridge pattern, Strategy pattern, and Visitor pattern are then measured against the value of performance metrics such as CPU usage, memory usage, and frame rate in the test application.

Table 6. The result of measuring the value of each performance metric after the Bridge pattern implementation

CPU Usage	Memory Usage	Frame Rate (*Frame per Second*)
49%	77%	55 fps

Referring Table 6 has generated measurements for each performance metric after the implementation of the Bridge pattern. The CPU usage value is 49% which is measured based on the total availability of the CPU that the application expends over a period of time. This value is smaller than the initial value of CPU usage before implementation, which is 74% which means that the Bridge pattern can reduce the CPU usage value or have a positive impact on performance. As for memory usage obtained by 77%, there was an increase from the initial value of memory usage before implementation, which was 75%. The increase in memory usage value has a negative impact on performance because it increases the need for resources by the application for memory allocation. For the frame rate value obtained at 55 fps, there is an increase from the initial value of the frame rate before implementation, which is 50 fps.

Table 7. The results of measuring the value of each performance metric after the implementation of the Strategy pattern

CPU Usage	Memory Usage	Frame Rate (*Frame per Second*)
41%	74%	58 fps

The measurement results after the implementation of the Strategy pattern can be seen in table 7. Table 7 has generated measurements for each performance metric after the implementation of the Strategy pattern. The CPU usage value is obtained at 41%. This value is smaller than the initial value of CPU usage before implementation, which is 74% which means that the Strategy pattern can reduce the value of CPU usage or have a positive impact on performance. As for the memory usage obtained by 74%, there was a decrease from the initial value of memory usage before implementation, which was 75% which had a positive impact on application memory usage. For the frame rate value obtained at 58 fps, there is an increase from the initial frame rate value before implementation, which is 50 fps which also has a positive impact on performance.

Table 8. The results of measuring the value of each performance metric after the implementation of the Visitor pattern

CPU Usage	Memory Usage	Frame Rate (*Frame per Second*)
48%	74%	57 fps

Table 8 shows the measurement results after the implementation of the Visitor pattern. The CPU usage value is obtained at 48%. This value is smaller than the initial value of CPU usage before implementation, which is 74% which has a positive impact on performance. As for memory usage obtained by 74%, there was a decrease from the initial value of memory usage before implementation, which was 75%. For the frame rate value obtained at 55 fps, there is an increase from the initial frame rate value before implementation, which is 50 fps which has a positive impact on performance.

3.1.2 Code smells Refactoring Results

The results of refactoring code smells contained in the source code of the test application are then measured against the value of performance metrics such as CPU usage, memory usage, and frame rate.

Table 9. The results of measuring the value of each performance metric after refactoring the code smells

CPU Usage	Memory Usage	Frame Rate (*Frame per Second*)
40%	73%	47 fps

Table 9 shows the measurement results for each performance metric after refactoring the code smells in the source code of the test application. Based on these results, the CPU usage value was obtained by 40%, this value was smaller than the initial value of CPU use before refactoring, which was 74%, which means that there is a decrease in CPU usage after refactoring. As for memory usage obtained by 73%, there was a decrease of 2% from the initial value of memory usage before implementation, which was 75%. For the frame rate value obtained at 47 fps, there is a decrease from the initial frame rate value before implementation, which is 50 fps. The decrease in fps value has a negative impact on performance.

Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

3.1.3 Design Principles Results

The results of the refactoring based on design principles are then measured against the value of performance metrics such as CPU usage, memory usage, and frame rate. The measurement results after refactoring based on the overall design principles can be seen in Table 10.

Table 10. The results of measuring the value of each performance metric after the implementation of the design principles.

CPU Usage	Memory Usage	Frame Rate (*Frame per Second*)
49%	77%	55 fps

Table 10 shows the measurement results for each performance metric after refactoring based on the design principles violated in the source code of the test application. Based on these results, the CPU usage value was obtained at 49%, this value was smaller than the initial value of CPU use before refactoring, which was 74%, which means that there is a decrease in CPU usage after refactoring. As for memory usage obtained by 77%, there was an increase from the initial value of memory usage before implementation, which was 75%. This results in a negative impact on the performance and memory usage of the application. For the frame rate value obtained at 55 fps, there is an increase from the initial value of the frame rate before implementation, which is 50 fps. The increase in fps value has a positive impact on performance.

3.2 Discussion

After measuring the value of performance metrics, the analysis of the test results after the implementation of the design pattern, namely the measurement results for each design pattern, found that the Strategy pattern has the smallest percentage of CPU use, which is 41% compared to the Bridge pattern and Visitor pattern which are 49% and 48%, respectively. The strategy pattern decreases CPU usage percentage by 44.6% which has a positive impact on application performance. For memory usage values, Strategy and Visitor patterns have a memory usage percentage of 74%, which means that they reduce the percentage of memory usage by 1% from the initial value before the implementation of the design pattern. In contrast to the Bridge pattern which has a memory usage percentage of 77%, which means it increases memory usage by 2% from the initial value of memory usage before the implementation of the design pattern. This happens because the Bridge pattern structure will add new objects to memory based on the class or function extracted into the new class. Therefore, bridge patterns have a negative impact on application memory usage. For frame rate values, Strategy pattern has a frame rate value of 58 fps, which means it increases the percentage of frame rate values by 31% from the initial value before the implementation of the design pattern. The bridge pattern and Visitor pattern have frame rate values of 55 fps and 57 fps, respectively, increasing the frame rate value percentages by 27.3% and 28.9%, respectively. This makes the strategy pattern implementation can increase the frame rate value better than the Bridge pattern and Visitor pattern.

Based on the results of measuring the value of the performance metric after refactoring code smells that have been carried out, the percentage of CPU usage is 40%. The results of the CPU usage measurement experienced a decrease of 34% compared to the initial value of CPU usage before implementation. As for memory usage, a percentage of 73% was obtained. This value is slightly smaller than the initial value of memory usage, which is 75%. For the frame rate value, a value of 47 fps is obtained, greater than the initial frame rate value of 50 fps. This negatively affects app performance because a decrease in fps values can lead to lagging of the app. Refactoring code smells can reduce CPU and memory usage, but the frame rate of the application actually increases. Although code smells are one of the symptoms of the appearance of design problems in the source code, not necessarily code smells can cause bugs or errors in the source code.

Based on the results of measuring performance metrics after refactoring based on design principles, the percentage of CPU usage was obtained at 49%. This value is smaller than the initial CPU usage value of 74%. The decrease in CPU usage value has a positive impact on performance. As for memory usage, a value of 77% was obtained, an increase from the initial value of memory usage before implementation. The increase in memory usage value has a negative impact on performance as well as memory. For the frame rate value, a value of 55 fps was obtained, there was a positive increase from the initial frame rate value of 50 fps. This result makes refactoring based on principle design can optimize CPU usage and frame rate, even if there is an increase in the memory usage value due to the addition of new objects in the memory.

Based on the results of the analysis of design patterns, refactoring code smells, and design principles, it was found that the largest decrease in CPU usage was obtained after refactoring code smells. As for memory usage, the results of refactoring code smells also provide the least value between design patterns and design principles. For the frame rate value, the highest increase is obtained after the implementation of the Strategy pattern. Overall, the application of a design pattern can affect the value of performance metrics depending on the design pattern used. Meanwhile, refactoring code smells can optimize CPU and memory usage, even if there is a decrease in the frame rate value. Refactoring based on design principles can optimize CPU usage and frame rates, although there is an increase in memory usage.

Building of Informatics, Technology and Science (BITS)
Volume 4, No 1, Juni 2022 Page: 226−234
ISSN 2684-8910 (media cetak)
ISSN 2685-3310 (media online)
DOI 10.47065/bits.v4i1.1739

4. CONCLUSION

The conclusion from the research that has been done is that the results obtained that the implementation of the design pattern has a different effect based on the design pattern used. Implementation of the Bridge pattern can increase the average memory usage by 2%, causing a negative impact on memory, although the Bridge pattern optimizes the average CPU usage value to 49% and frame rate to 55 fps. The increase in average memory usage in the Bridge pattern is due to the addition of new objects stored in memory based on the extraction of the refactored class or function. Meanwhile, the Strategy pattern and Visitor pattern provide positive changes to each of the performance metric values. The results of the refactoring measurement of code smells resulted in optimal changes in the average value of CPU and memory usage, namely 40% and 73% which had a positive impact compared to using design patterns and design principles, although there was a decrease in the frame rate value to 47 fps which had an impact on negative on performance. While the measurement results after the implementation of design principles as a whole, it was found that design principles can optimize CPU usage to 49% and frame rate to 55 fps which has a positive impact on performance, although there is an increase in memory usage by 77% which has a positive impact on performance. Based on these results, getting optimal performance results for each performance metric, it can be done by implementing the Strategy pattern and Visitor pattern. Meanwhile, to get the least CPU usage can be obtained by refactoring the code smells contained in the source code. The implementation of the design principle has not been effective compared to the implementation of design patterns and refactoring of code smells. Further research can be continued using other performance metric values to determine the impact of the design pattern on these metric values and can use other strategies or approaches.

REFERENCES

[1] A. Ahmad, C. Feng, M. Tao, A. Yousif, and S. Ge, “Challenges of Mobile Applications Development: Initial Results,” IEEE Access, vol. 6, pp. 17711–17728, 2018, doi: 10.1109/ACCESS.2018.2818724.
[2] A. A. Saifan and A. Al-Rabadi, “Evaluating maintainability of android applications,” in ICIT 2017 - 8th International Conference on Information Technology, Proceedings, Oct. 2017, pp. 518–523. doi: 10.1109/ICITECH.2017.8080052.
[3] A. Qasim, A. Munawar, J. Hassan, and A. Khalid, “Evaluating the Impact of Design Pattern Usage on Energy Consumption of Applications for Mobile Platform,” Applied Computer Systems, vol. 26, no. 1, pp. 1–11, May 2021, doi: 10.2478/acss-2021-0001.
[4] P. K. Aggarwal, P. S. Grover, and L. Ahuja, “A Performance Evaluation Model for Mobile Applications,” in 2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU), Apr. 2019, pp. 1–3. doi: 10.1109/IoT-SIU.2019.8777497.
[5] A. Vishnyakov and S. Orlov, “Software architecture and detailed design evaluation,” in Procedia Computer Science, 2015, vol. 43, no. C, pp. 41–52. doi: 10.1016/j.procs.2014.12.007.
[6] A. S. Cairo, G. de F. Carneiro, and M. P. Monteiro, “The impact of code smells on software bugs: A systematic literature review,” Information (Switzerland), vol. 9, no. 11, Nov. 2018, doi: 10.3390/info9110273.
[7] H. Singh and S. Imtiyaz Hassan, “Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment,” International Journal of Scientific & Engineering Research, vol. 6, no. 4, pp. 1321–1324, Apr. 2015, [Online]. Available: http://www.ijser.org
[8] M. Mahendra and B. Anggorojati, “Evaluating the performance of Android based Cross-Platform App Development Frameworks,” in ACM International Conference Proceeding Series, Nov. 2020, pp. 32–37. doi: 10.1145/3442555.3442561.
[9] G. Hecht, N. Moha, and R. Rouvoy, “An empirical study of the performance impacts of android code smells,” in Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016, May 2016, pp. 59–69. doi: 10.1145/2897073.2897100.
[10] M. Willocx, J. Vossaert, and V. Naessens, “Comparing performance parameters of mobile app development strategies,” in Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016, May 2016, pp. 38–47. doi: 10.1145/2897073.2897092.
[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Pattern: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH, 1995.
[12] A. Shvets, Dive Into Design Patterns. 2021.
[13] L. Sousa et al., “How Do Software Developers Identify Design Problems?: A Qualitative Analysis,” in ACM International Conference Proceeding Series, Sep. 2017, pp. 54–63. doi: 10.1145/3131151.3131168.
[14] F. A. Fontana, V. Ferme, and M. Zanoni, “Towards Assessing Software Architecture Quality by Exploiting Code Smell Relations,” in Proceedings - 2nd International Workshop on Software Architecture and Metrics, SAM 2015, Jul. 2015, pp. 1–7. doi: 10.1109/SAM.2015.8.
[15] A. Shvets, Dive Into Refactoring. 2019.
[16] K. Kandt, “Software Design Principles and Practices,” 2003.
[17] R. C. Martin, “Design Principles and Design Patterns,” 2000. [Online]. Available: www.objectmentor.com
[18] M. Jaiswal, “Software Architecture and Software Design,” International Research Journal of Engineering and Technology (IRJET) e-ISSN, pp. 2395–0056, 2019, [Online]. Available: https://ssrn.com/abstract=3948301
[19] T. Byambaa, “Open source flutter apps.” Accessed: May 16, 2022. [Online]. Available: https://github.com/tortuvshin/open-source-flutter-apps
[20] S. Guo, “GSYGithubApp.” Accessed: May 16, 2022. [Online]. Available: https://github.com/CarGuo/gsy_github_app_flutter