IPTEK The Journal of Technology and Science, 35(1), 2024 (e/pISSN:2088-2033, 0853-4098)

DOI: 10.12962/j20882033.v35i1.16585 | Received 11 April, 2023; Revised 16 August, 2023; Accepted 12 Sept, 2023

ORIGINAL RESEARCH

INVESTIGATING DESIGN PATTERNS IMPACT ON APPLICATION PERFORMANCE AND COMPLEXITY

Siti Rochimah*[1] | Rizky Januar Akbar[1] | Kholed Langsari[2]

[1]Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya, 60111, Indonesia

[2]Department of Computer Science, Fatoni University, Khao Tum, 94160, Thailand, 60111, Thailand

Correspondence

*Siti Rochimah, Dept of Informatics , Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia. Email: siti@if.its.ac.id

Present Address

Gedung Teknik Informatika, Jl. Teknik Kimia, Kampus ITS Sukolilo, Surabaya 60111, Indonesia

Abstract

Many studies in the literature have a premise that design patterns improve the quality of object-oriented software systems. Considerable research has been devoted to re-designing the system to improve software quality, mainly on its maintainability and reliability. Less attention has been paid to evaluating the impact of the performance efficiency quality factor. This research investigates the impact of design patterns on application performance and complexity. It is, therefore, beneficial to evaluate whether the design patterns may improve its performance and complexity or even decrease it. The research demonstrates scientific evidence in quantitative values through experimentation on a case study to present its influences. This paper uses an object-oriented enterprise project named SIA as a case study. Some issues related to design patterns are addressed. The selection of the design pattern is based on the application context issue. Three attributes related to performance efficiency are evaluated: time behavior, resource utilization, and capacity measures. The complexity is also evaluated. We use Apache JMeter and Java Mission Control as tools to support experimentation. The experiment results show that design patterns may decrease the quality of time behavior and resource utilization whilst they may increase the quality of capacity measures and complexity to a significant degree.

KEYWORDS:

Design Patterns, Performance Efficiency, Complexity, Software Quality, Software Evolution

1 | INTRODUCTION

The durability of concrete plays a critical role in controlling its serviceability. Buildings in direct contact with water, such as marine infrastructure, should resist water penetration so the concrete's reinforcements do not experience corrosion. Corrosion is an electrochemical process affected by the availability of water and oxygen. Therefore, it is generally accepted that concrete durability, which is often associated with corrosion, is affected by concrete pore structure. Moreover, the concrete designing and

development of software applications have never been an easy task. The process is often time-consuming and requires interaction between several different aspects. Enterprise software developers usually try to develop enterprise software applications that satisfy business needs and achieve high-quality software within a short development time. Even the most complex systems are built by using smaller parts. Such parts may be built by using even smaller parts. They need to communicate with each other to perform the whole function of a system. The object-oriented approach attempts to manage the system's complexity by abstracting out knowledge and encapsulating it within interacting objects.

A software design pattern is usually utilized to identify reoccurring design problems. The concept of design patterns has been presented in software engineering for a relatively long time[1]. A design pattern is an abstraction of practical experience and empirical knowledge, describing a problem it addresses and a solution. Design patterns have many functions in software engineering along with other pattern categories, for instance, reengineering patterns[2] or analysis patterns[3]. Design patterns facilitate design and development by expressing ideas and solutions in a high-level language. Patterns may improve software quality by making it more reusable, maintainable, and performance efficient. Developers are increasingly more aware of how and when to use different patterns.

Design patterns must be applied with caution. They provide solutions at a certain level of granularity, often the class level, in a standard object-oriented language. The problems are often centered around how to encapsulate variability in a design. Patterns, when implemented, often come with the cost of an extra layer of indirection, creating the way for increased abstraction in the program. In addition, there are design patterns that can reduce object calls and layers, such as Singleton. It creates single-handle objects and calls through them; it directly improves the application's performance by reducing traditional object-oriented design by creating extra layers and providing design flexibility.

On the other hand, applying several design patterns might create several layers of indirection, which may positively or negatively impact performance. Design patterns provide discipline in creating or refactoring a better software structure but cannot guarantee software quality. The true benefit is only realized if a given collection of design patterns is used regularly in a specific domain and context. Those several layers of indirection may reduce the application's performance.

Performance is one of software quality's important and essential attributes[4]. Software clients usually regard performance as an important standard when deciding whether the software is good. Usually, the client doesn't have to know what design decisions were made. However, knowing how the software performs and responds is more important. The performance of software reflects its efficiency because the software that properly uses resources usually responds fast. Performance is an important quality attribute, which can be measured as throughput and resource utilization. However, performance is only one of many parameters that determine the quality of the final product. Performance-related aspects can be categorized and characterized by time behavior, resource utilization, and capacity compliance[5].

A survey of design patterns' impact on software quality[4][6] has shown that performance is one of the quality attributes affecting software systems' quality. However, the number of studies and the coverage of the addressed patterns are insufficient to conclude their impact on performance. Many studies have the premise that design patterns improve the quality of object-oriented software systems. Some studies suggest that using design patterns does not always result in an appropriate design[6]. Yet, less attention has been paid to measuring the impact of performance efficiency by using design patterns. Thus, there are remaining questions such as "Do design patterns impact application performance?" and "If it has an impact, is it a negative or positive impact on performance?".

The main idea of this paper is to investigate the impact of design patterns on application performance and its complexity. This paper uses an enterprise project named SIA as a case study. The SIA project is a particular academic information system that our university utilizes, and it is designed under the Java EE platform[7][8]. Some issues related to design patterns are addressed. We use ISO/IEC 25010[9] and ISO/IEC 25023 (SQuaRE)[10] as standards to evaluate the experiment results. The SQuaRE international standard aims to define quality measures by quantitatively measuring system and software product quality in terms of characteristics and sub-characteristics. In this work, we focus on implementing design patterns, especially the "Gang of Four" (GoF) design patterns, through the refactoring process to see the impact result in terms of performance efficiency and complexity.

The paper is organized as follows: Section 1 describes the background and problem statement of the research. Section 2 explains the previous research, followed by the research methods in Section 3. Next, Section 4 describes the case study used for the

experimentation. Section 5 describes the detailed steps of the experimentation, followed by the results described in Section 6. Finally, Section 7 concludes the overall research findings.

2 | PREVIOUS RESEARCHES

Rochimah et al.[11] compares software quality before and after implementing the design pattern. They also use SIA as a case study. The results show that the quality of applications that use design patterns increases significantly. Next, the same author also examines the impact of implementing design patterns on software quality, especially in the maintainability aspect[12]. Experimental results show an increase in the quality of the maintainability aspect but do not impact the analyzability and testability aspects.

Rochimah et al.[13] researched the relationship between the software architecture in which the design pattern was applied and maintainability quality. Maintainability measurement is done by using CK metrics. The result is that the design pattern application has been proven to improve the software architecture quality and maintainability significantly.

Nazar et al.[14] examines ways to detect the existence of a design pattern in a source code through the use of features. They used the feature as the entrance to their detection process. They made a design pattern detector using 15 source code features to detect design patterns. However, the experimental results regarding recall and precision are still not high enough.

3 | METHOD

Our approach is structured in three fundamental phases: i.e., analysis, refactoring, and performance measurement, as depicted in Figure 1 . Descriptions for each detailed process are as follows.

3.1 | Phase 1: Analizing

In the first phase, we analyze the existing system to obtain the complexity of the application system. The analysis consists of four main processes, i.e., reverse engineering, measuring complexity, identifying the problem, and design pattern selection.

In the first process, reverse engineering, we use the object-oriented reverse engineering technique[2], focusing on analyzing the subject system to identify the system's components and their interrelationships. Next, we create representations of the system at a higher level of abstraction. We use several sources of information while doing reverse engineering, such as reading the existing documentation and the source code, running the software, and using tools to generate high-level views of the source code. These sources of information help a lot in analyzing, documenting, and identifying potential application problems. The result of this activity is a detailed system in terms of Architecture View and Class Diagram.

In the second process, measuring the complexity of the application, we utilize OO metrics[15] to measure the complexity of existing SIA source code. The measurement results are values regarding SIA complexity, especially in terms of Total Lines of Code (TLOC), Number of Classes (NOC), Number of Methods (NOM), Number of Packages (NOP), and, of course, the values of McCabe Cyclomatic Complexity (MCC).

In the third process, problem identification, we use the results of those previous processes to identify potential problems that may occur in the application.

In the fourth process, design patterns selection, the results of previous processes added with the identified potential problems give the sign or indication of code and design smell issues[16] [16]related to existing SIA. The design pattern is selected based on GoF design pattern categories. GoF classified design patterns' purpose into creational, structural, and behavioral. The design pattern is selected based on the problem faced in some specific contexts and is judged by experts or researchers. The result of this activity is the suitably selected design pattern that will be adapted and implemented into the SIA in the next refactoring process.

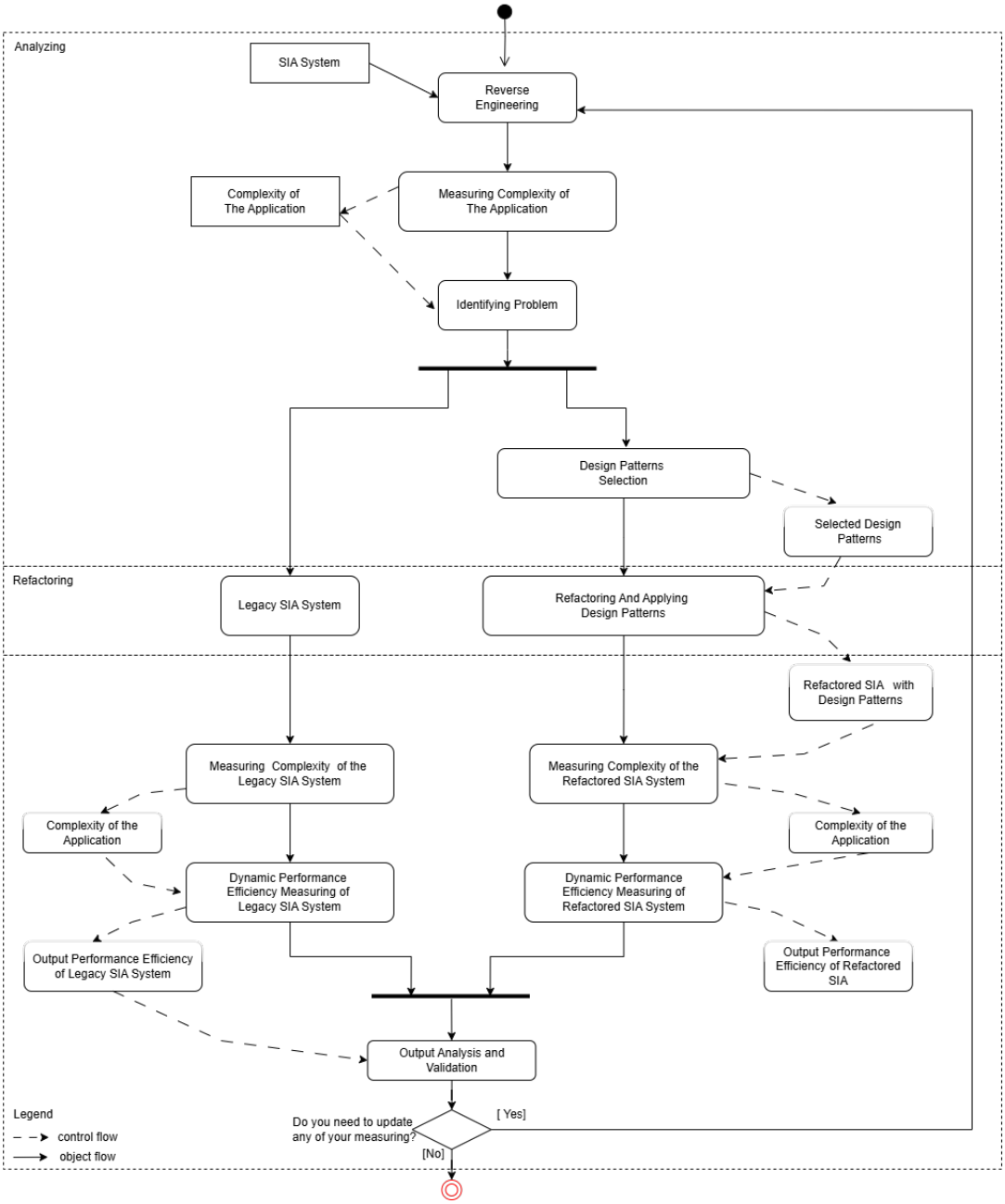

FIGURE 1 The Proposed Approach

3.2 | Phase 2: Refactoring

Refactoring involves changing activities over the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. The process involves the removal of duplication, the simplification of complex logic, and the clarification of existing code. When we refactor, we relentlessly restructure and modify the code to improve its design. The main activity is to apply design patterns to existing systems through refactoring techniques. In this activity, we follow the IMPACT refactoring process model[17]. There are four fundamental steps, i.e., identify and mark refactoring candidates, plan the activities, act on the planned refactoring tasks, and test to ensure behavior preservation. Figure 2 gives a UML activity diagram of the refactoring process. A detailed description of each step is below.

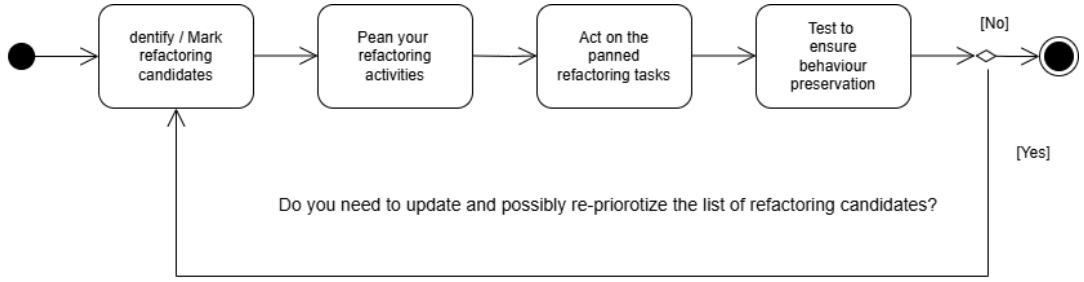

FIGURE 2 Activities of IMPACT refactoring process model[17]

The first step, identifying and marking refactoring candidates, analyzes and identifies the code. This process is carried out through manual code and design review to find smells and determine candidates for refactoring. Manual reviews are more effective and less error-prone because they can more effectively consider and exploit domain knowledge, the context of the design, and design expertise.

The second step, planning the refactoring activities, identifies smells, analyzes their impact, prioritizes them, and prepares a plan to address them. To analyze the impact of a smell, we consider factors such as severity, scope, and interdependence. After analyzing the impact, we prioritize them based on the following factors: potential gain after removing the smell, available time, and test availability for the target modules. Based on the identified smells prioritized list and their refactoring, the design pattern selection process starts to select a suitable design. Next, an execution plan for the refactoring can be appropriately formulated.

The third step, acting on the planned refactoring tasks, may take up planned refactoring tasks and execute them by carrying out the refactoring on the code. In this process, we use automated refactoring tool support provided by IDEs to carry out the refactoring tasks.

The fourth step, testing to ensure behavior preservation, is crucial in refactoring. The refactoring activity should be followed by automated regression tests to ensure that the behavior remains unchanged after the refactoring. We first test for the entity that needs to be refactored, then refactor it, and finally test it to verify the behavior.

The refactoring process produces a refactored SIA with a properly applied design pattern. On the other hand, we do nothing to the existing SIA. The existing system remains the same for comparison. The refactored and existing SIAs are the outputs of this process and are used in the next step of the approach.

3.3 | Phase 3: Measuring

This last phase evaluates the impact of design patterns on application performance and complexity. It consists of two main activities, i.e., measurement and analysis-validation of the result. In this first activity, we perform both static and dynamic measurements. In static measurement, we measure various parameters of the source code to get its complexity and other important aspects. In dynamic measurement, we measure the parameters of the executable module. The module is activated first; then, during its execution, its behaviors are captured and measured in several variable values. A detailed description of each step is below.

The static measurement measures the complexity of the application. The static measurement produces summaries and details of both refactored and existing SIA source codes. The metrics that are measured are NOP, NOC, NOM, TLOC, MCC, Number of Interfaces (NOI), Number of Attributes (NOF), Weighted Methods per Class (WMC), Lack of Cohesion of Methods (LCOM), Efferent Coupling (CE), and Afferent Coupling (CA). We also compare the difference between refactored and existing SIA to determine the changed point and internal structure that have been refactored.

The dynamic measurement measures performance efficiency. The dynamic measurement is conducted on the executable module of refactored and existing SIA. We utilize core performance testing activities defined by Meier. There are seven core activities, i.e. (i) identify test environment; (ii) identify performance acceptance criteria; (iii) plan and design tests; (iv) configure test environment; (v) implement test design; (vi) execute tests; and (vii) analyze, report, and retest.

We measure three performance efficiency factors: time behavior, resource utilization, and capacity conformance. To achieve this target, we use performance measurement and profiling tools. We use Apache JMeter to measure time behavior and performance efficiency compliance using load test functional behavior. The JMeter tool sends several requests that simulate user activities on the application.

We use Java Mission Control (JMC) and Java Profiling tools to measure the resources used in a specific period of time. By collecting information on response time, content, and resource usage, we can calculate the measurement of all defined parameters. The test data is generated in CSV and XML file format. These tools allow us to measure performance efficiency according to specified criteria.

The test scenarios utilized for performance efficiency measurement should reflect the main functionality of user activity. The test activities must represent the following data operation: querying, creation, removal, and update. Data querying is the most prevalent activity. The test scenarios include activities typical for this kind of application: (i) student addition, (ii) listing student details, (iii) student removal, and (iv) report generation.

During each test, we capture several performance efficiency parameters. We consider parameters directly related to application performance and supported by the tools. Those parameters are grouped into time behavior, resource utilization, and capacity conformance.

The first category, time behavior, consists of three parameters: mean response time, response time, and throughput conformance. The mean response time measures the time the system takes to respond to a user action. Response time conformance measures how well the system's response time meets the specified target. Throughput conformance measures how well the throughput meets the specified targets.

The second category, resource utilization, consists of mean processor and memory utilization. The mean processor utilization measures how much time is used to execute a given set of tasks compared to the operation time. The mean memory utilization measures how much memory is used to execute a given set of tasks compared to the available memory.

The third category, capacity conformance, consists of transaction processing capacity conformance. This parameter measures how many concurrent transactions can be processed at any given time against the specified target.

The second activity is output analysis and validation. This activity analyzes and validates the performance efficiency measurement results of existing and refactored SIA systems. The result of this final step provides a data source for analyzing and investigating the impact of design patterns during the refactoring process. Based on performance measurement criteria, the results determine the impact of the design pattern, whether it has a positive or negative effect.

4 | THE CASE STUDY

We use the SIA project as a case study. It is the academic information system utilized by Institut Teknologi Sepuluh Nopember. We create a synthetic version of SIA systems for research purposes. The synthetic version consists of six modules: sia-domain, sia-learning, sia-assessment, sia-curriculum, sia-equivalency, and SIA-Framework. Each module has its functionalities. The SIA-Frameworkwork is a central module. The sia-Domain is a linker of other modules to the database. Four other modules perform academic functions. The architecture is shown in Figure 3 .

SIA was created using Java Enterprise Edition technology with Hierarchical Model-View-Controller architecture, Spring MVC, and Hibernate the ORM framework as the libraries. The pattern decomposes the client tier into a hierarchy of parent-child MVC layers. The repetitive application of this pattern allows structured-client-tier architecture to reduce dependencies and increase extensibility. SIA uses Eclipse Virgo and OSGi Framework. The backend web server that is used to run SIA is Apache Tomcat. SIA uses PostgreSQL as the database.

The SIA framework is created based on the layers of modularity architecture, which serves the principle of separation of concerns. There are sia-web, sia-service, sia-data, sia-plugin, and sia-domain layer. All these layers are built and deployed independently on the OSGi framework. Figure 4 illustrates the UML component diagram of the SIA system.

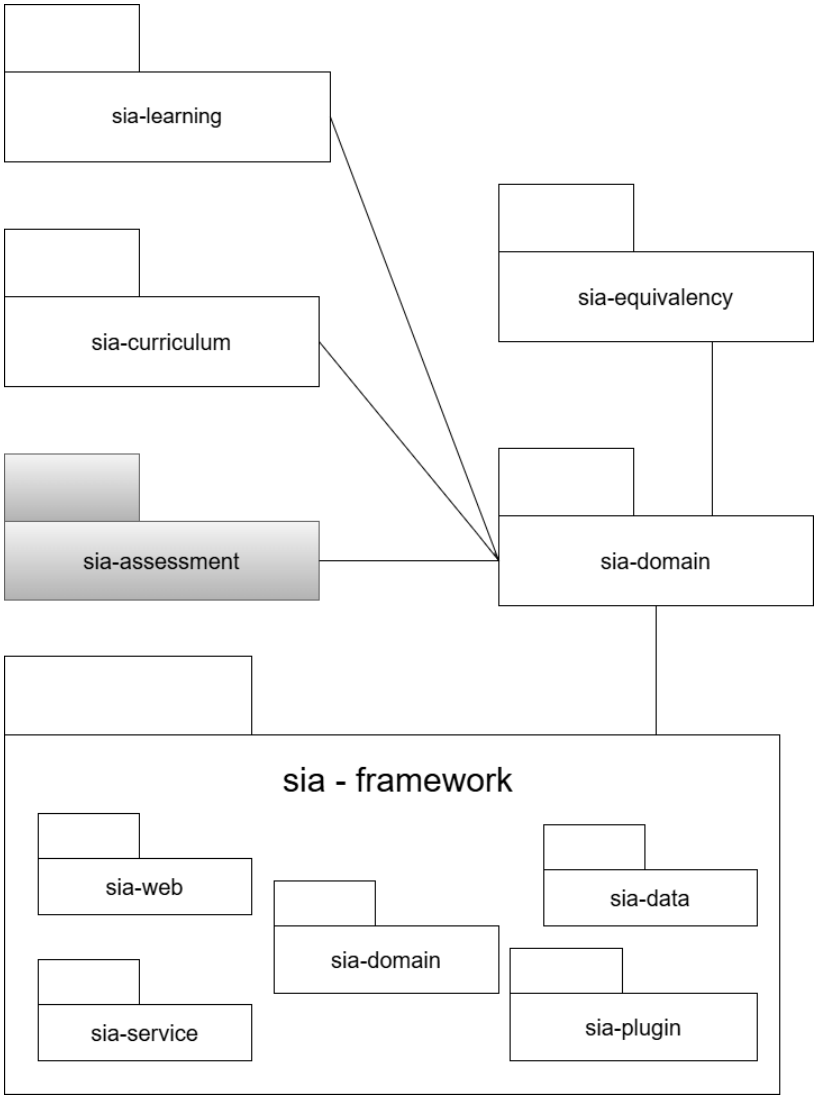

FIGURE 3 Architecture of SIA

5 | THE EXPERIMENTATION

In our experimentation, sia-assessment is chosen as chosen, which is modified by designing patterns. This module allows authorized users to submit or change assignments and examination marks, final grades, reports, generate student transcripts, produce grade point average scores, etc.

This module serves up to 14 services and consists of three main packages: Controller, Service, and Repository. The controller acts as an interface between the model and view components. The service is responsible for the presentation and data storage middle layer. The Repository separates the logic that retrieves data and maps it into an entity model from the business logic. The controller consists of five main classes, the Repository of 18 classes and 18 interfaces, and the Service of 24 classes and 18 interfaces.

After some observations on the assessment module, it is identified that the Controller should only be responsible for preparing data for the model and mapping it into a view name. Yet, it can also directly respond to and complete the request. Most classes in Controller make direct calls into classes in Service. This situation creates one-to-many dependencies, making communication between both packages more complicated, as depicted in Figure 5 .

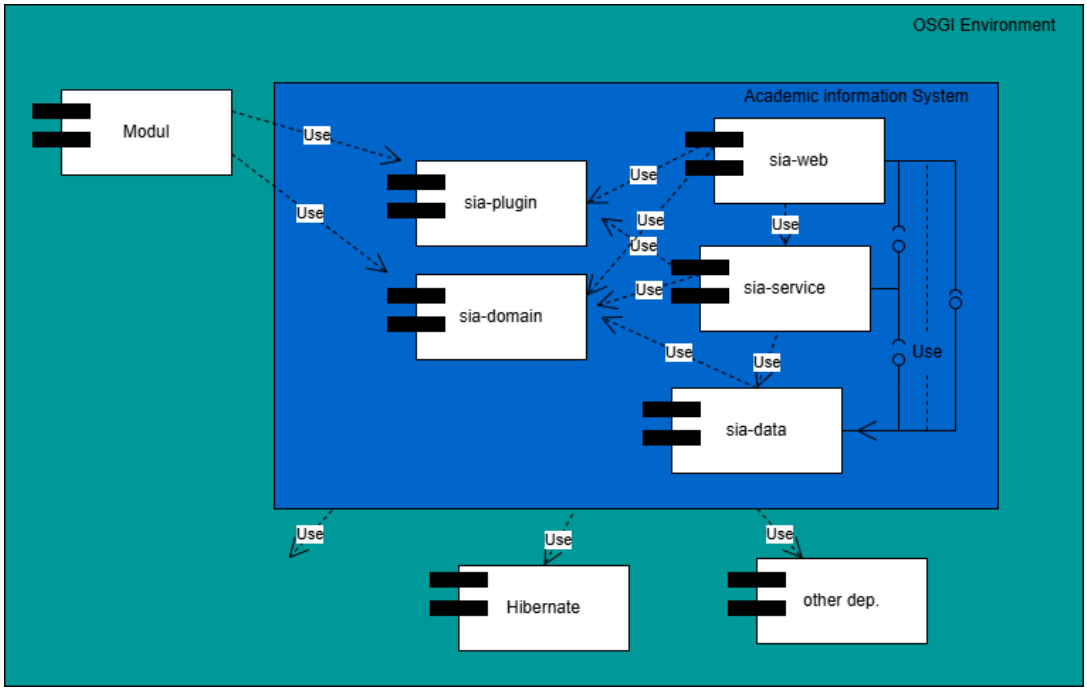

FIGURE 4 Component Diagram of SIA

Based on the problem, refactor the code using the arcade pattern. Theoretically, this pattern may improve the independence of the classes, making it easy to change or even replace them without impacting outside code. Facade addresses the problem by forcing programmers to use classes indirectly through a well-defined single point of access, thereby shielding the programmers from the complexity of the code on the other side of the facade. Figure 6 shows package dependencies after implementing the facade.

Facade implementation between Controller and Service aims to decrease coupling and communication complexity between the two packages, although the consequence is increasing the number of packages. Usually called an extra layer of indirection.

6 | THE MEASUREMENT RESULTS

According to our approach, two types of measurement are to be conducted: sta:d dynamics, which involves 11 metrics of this static aspect and six metrics of the dynamic aspect for bathing and the refactored systems.

6.1 | Results of Static Measurement: The Complexity

The complexity results are grouped into two categories, i.e. (i) metric values, which are obtained from the total number of artifacts in the source code, and (ii) metric values, which are related to the complexity. Increasing the number of artifacts is not always correlated with increasing maintenance effort since many other parameters determine the level of maintenance. Moreover, the higher the program's complexity, the more difficult it is to maintain.

Table 1 shows the results of the total number of artifacts in both existing and refactored systemArtifactesults are also depicted by the chart in Figure 7 . It is worth noting that applying design patterns usually leads to increased artifacts. This can not be avoided. However, the increasing number of artifacts is not always followed by increased complexity. There is no definite correlation between the increasing number of artifacts and the increasing complexity of the program. The more decisive is the internal structure of the program itself.

Table 2 and Figure 8 show the mean metric values related to the complexity of existing and refactored systems. It is clear that the value of LCOM, CE, and CA, which reflect program dependency, dramatically increased. The value of these three metrics

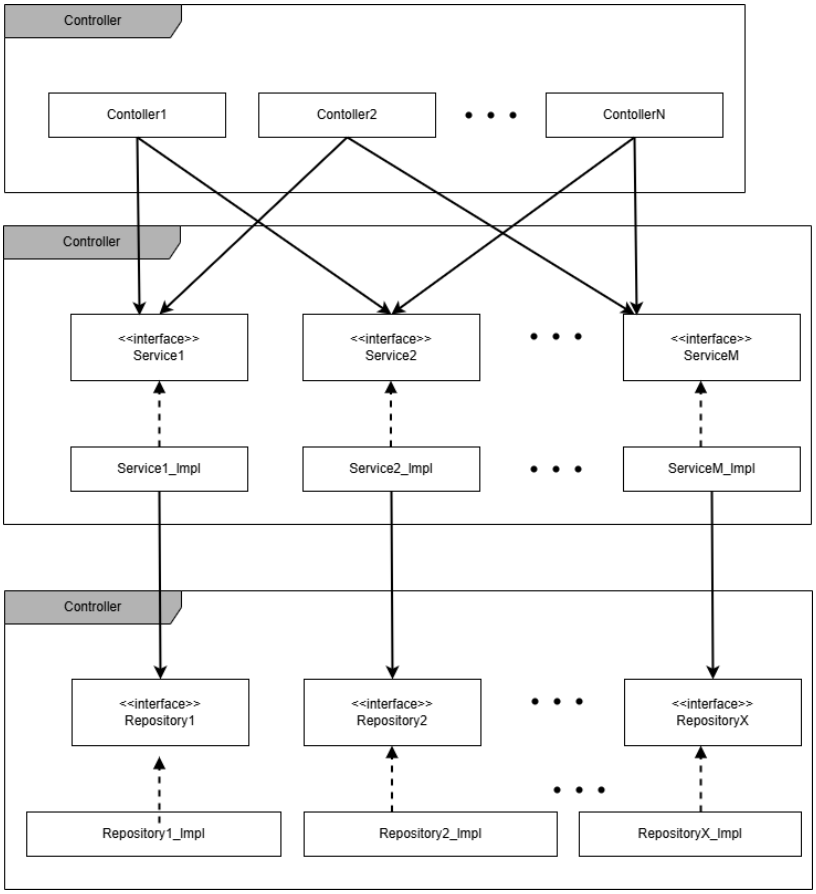

FIGURE 5 Dependencies and Communication among Packages

TABLE 1 Total Number of Artifacts

Version	NOP	NOC	NOI	NOF	NOM	TLOC (KLOC)
Existing	3	47	36	79	250	3117
Refactored	3	48	36	82	261	3427

greatly determines the quality of the maintenance level. Lower LCOM, CE, and CA values indicate that the program becomes simpler, more independent, and easier to maintain. A slight increase in VG and WMC is inherently caused by the increasing number of artifacts, especially in the Facade package.

TABLE 2 Metric Value Related to the Complexity

Version	Mean of				
	VG	WMC	LOCM	CE	CA
Existing	1.268	6.745	4.968	25.667	7.667
Refactored	1.598	7.253	3.891	19.891	5.376

6.2 | Results of Dynamic Measurement: The Performance Efficiency

In dynamic measurement, we conduct four scenarios: SC1: add new users; SC2: list student details; SC3: remove users; and SC4: generate a report. For each scenario, we simulate it using 10 concurrent users for the first iteration, which is increased by

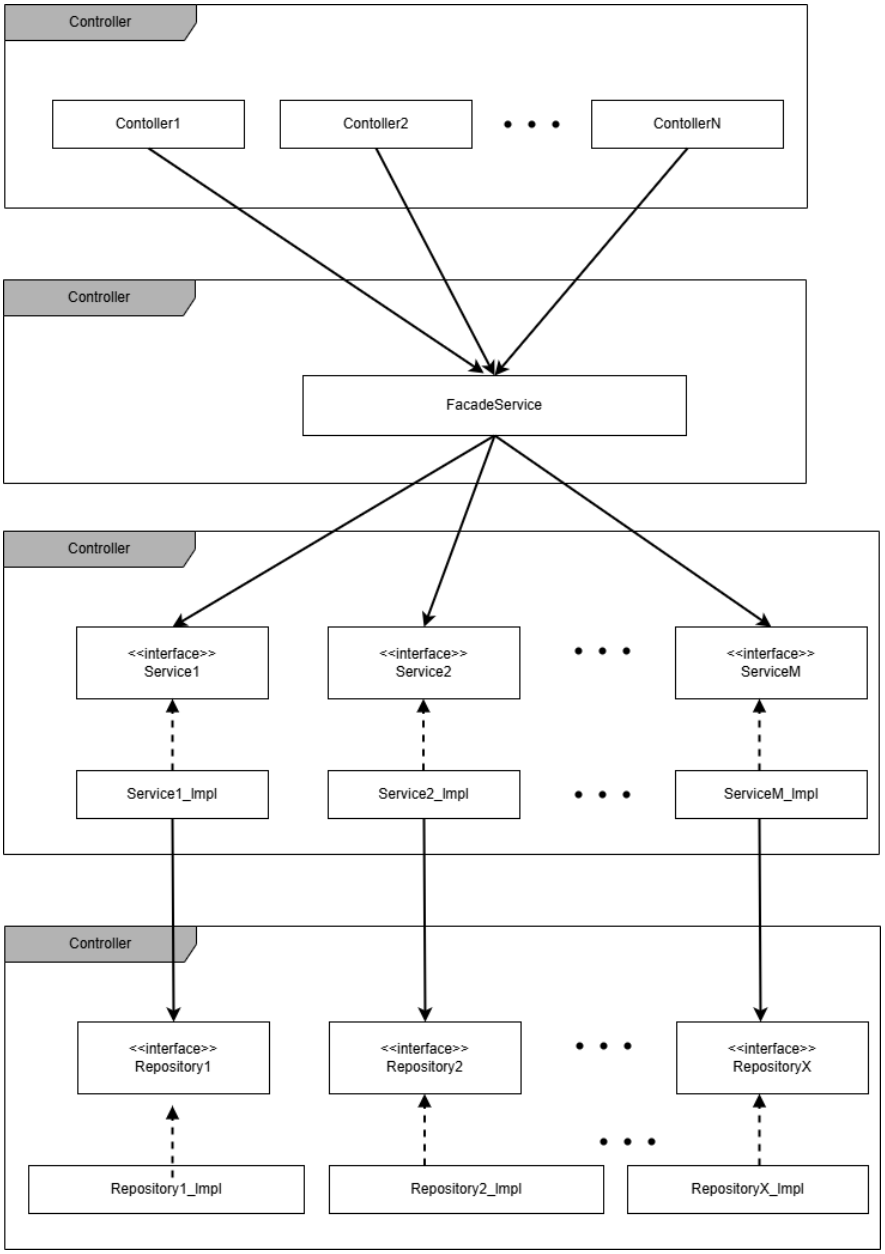

FIGURE 6 Package Dependencies after Facade Implementation

10 for the next iteration until there are 3 0 users. We conduct the test three times for each iteration to get reliable and meaningful results.

Performance measurement is conducted by setting 1000 threads as target load, 30 minutes Ramp-Up Time, 100 Ramp-Up Steps, and 10 minutes holding time to reach the tar et load. Starting from 10 users as an initial thread, they are increased by 10 users every 0.3 minutes until 1000 users are reached. When the target load is reached, the threads continue hitting the server for 10 minutes before being halted. The results are shown in the subsequent sections.

6.2.1 | Mean Response Time

The mean response time of the existing and refactored SIA is shown in Table 3 and illustrated in Figure 9 . The mean response time for each iteration is obtained by averaging the results of three executions. The response time measure is in the second (sec)

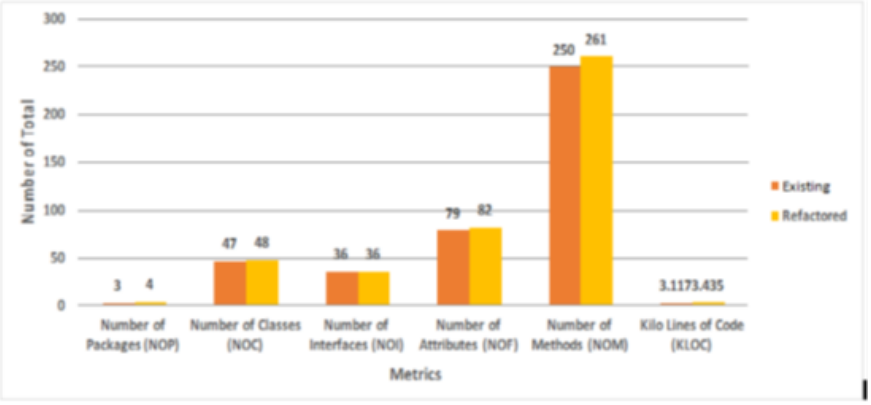

FIGURE 7 The chart of the total number of artifacts.

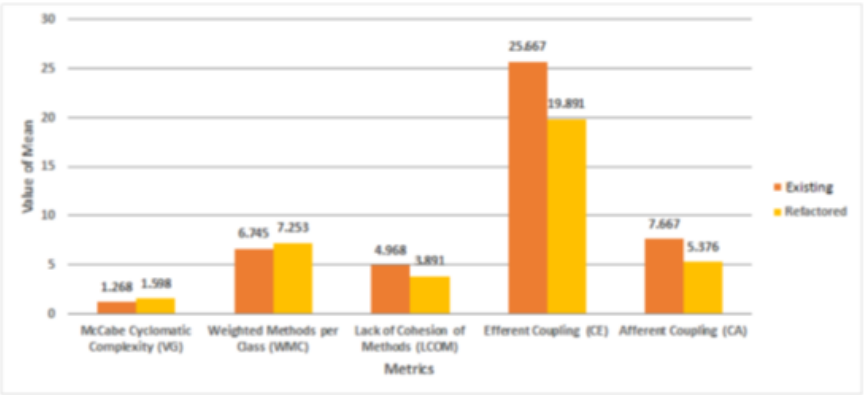

FIGURE 8 Chart of the Complexity Values

unit. Figure 9 shows that the average response time for the existing system is less than for the refactored system. Out of four scenarios, only the second scenario of the existing system exceeds the response time, and even then, it is only very small and adrift compared to the refactored system. The existing system has a faster response time for the other three scenarios than the refactoring system. The experiment results show that the average response time for the existing system is 36,51 seconds, while for the refactored system is 40,51 seconds.

This is very reasonable considering that a facade layer is added in the refactored system, which affects response time because there is an extra layer of independent distribution. This section concludes that the application of design patterns generally increases response time.

TABLE 3 Mean Response Time

Version		Response Time (sec)			
		Iteration1	**Iteration2**	**Iteration3**	**Iteration4**
Existing	SC1	42.68	41.07	44.69	42.81
	SC2	31.69	29.26	17.89	26.28
	SC3	17.93	14.97	16.03	16.31
	SC4	61.70	64.11	56.05	60.62
Refactored	SC1	51.25	45.86	50.28	49.13
	SC2	31.89	28.93	16.93	28.92
	SC3	19.64	19.23	17.11	18.33
	SC4	80.71	67.14	58.08	68.64

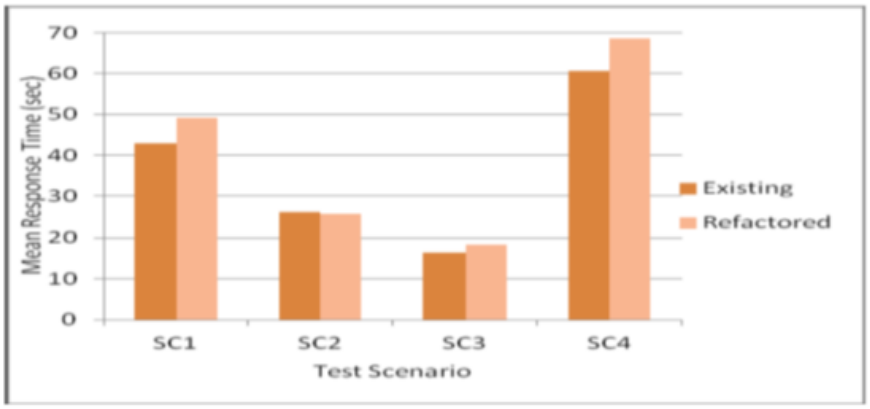

FIGURE 9 Chart of the Mean Response Time

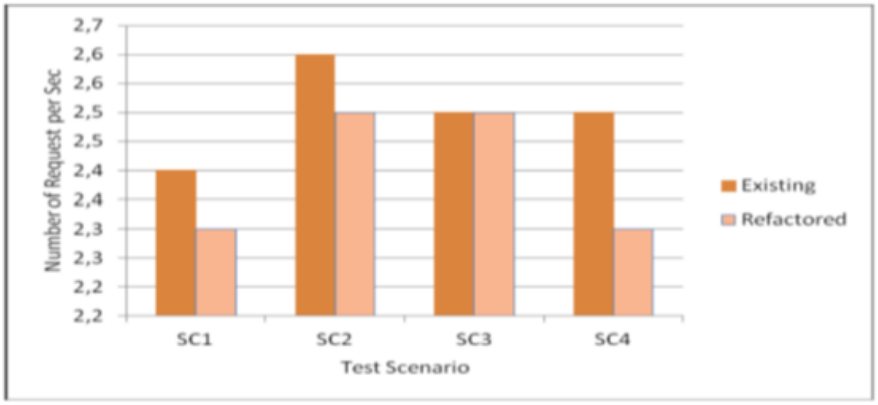

FIGURE 10 Chart of the Throughput Conformance

6.2.2 | Response Time Conformance

Response time conformance is calculated as the ratio of the response time of the refactored and the existing system. This value should be reached as low as possible to get a better system, usually less than 1. Table 4 shows that the values of SC1, SC3, and SC4 are greater than 1, which means that the refactored system's response time is worse than the existing system's. Only SC2 shows a value of less than 1, yet it is close to 1.

The conclusion of this section is similar to the previous one. The reason is similar: the extra layer of indirection of the facade layer.

6.2.3 | Throughput Conformance

Throughput is formulated as the number of requests per unit time. Time is calculated as the starting time of the first request and the last request's ending time, including the pauses or intervals between requests, which are considered a time for server load. Table 5 and Figure 10 show the throughput of the existing and refactored systems.

Based on those values, the throughput conformance is calculated as depicted in Table ftab5. The throughput conformance values are greater or equal to 1, which means that the throughput of the existing system is better than the refact red one. Using the same reason as the previous ones, this is due to the extra layer of indirection, which increases the execution time.

TABLE 4 The response time conformace.

Version	SC1	SC2	SC3	SC4
Existing	42.81	26.28	16.31	60.62
Refactored	49.13	25.92	18.33	68.64
Response Time Conformace	1.15	0.99	1.112	1.13

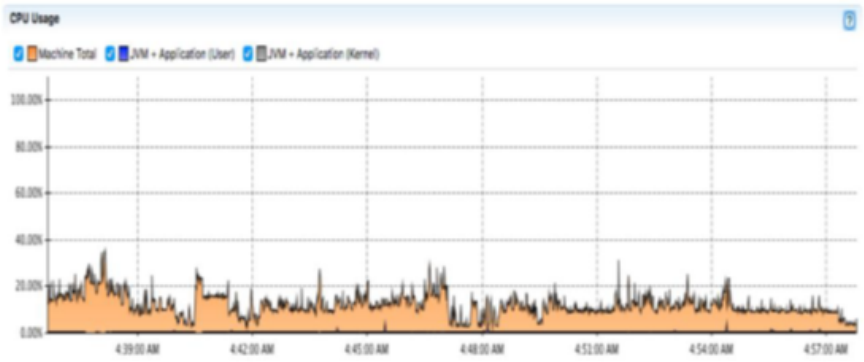

FIGURE 11 CPU Usage of Existing System

TABLE 5 Throughput Conformance

Version	SC1	SC2	SC3	SC4	Total
Existing (# request/sec)	2.400	2.600	2.500	2.500	10.000
Refactored(# request/sec)	2.300	2.500	2.500	2.300	9.600
Throughput Conformance	1.043	1.040	1.000	1.087	1.042

6.2.4 | Mean Processor Utilization

We use CPU utilization to investigate system performance, time, and the percentage of CPU processes utilized when executing a task. These data may be used to track any CPU performance regression or improvement, making them useful for investigating performance problems.

Figure 11 and Figure 12 show the percentage CPU usage of the systems. There is little difference in CPU usage between the existing and the refactored systems. The results show that the average CPU usage is 15% for the existing system and 17% for the refactored system. The hot thread is reached for the Main function, which is 37.88% for the existing system and 39.61% for the refactored systems. Also, the Local Descriptor Scanner function reaches 16.16% for the existing system, while the refactored system reaches 16.62% of CPU usage.

This section concludes that the refactored system requires more CPU utilization than the existing system. This is due to the extra Facade layer of indirection, which increases the number of artifacts and the CPU process allocation.

6.2.5 | Mean Memory Utilization

The memory utilization function on JMC allows us to check the memory allocation rate in the running application. The experiment results show that the refactored system uses an average of 574.79 megabytes of allocated memory for TLABs (Thread Local Allocation Buffer), while the existing system uses 572.98 megabytes. The refactored system uses more memory due to the extra Facade layer of indirection, which increases the number of artifacts and memory allocation.

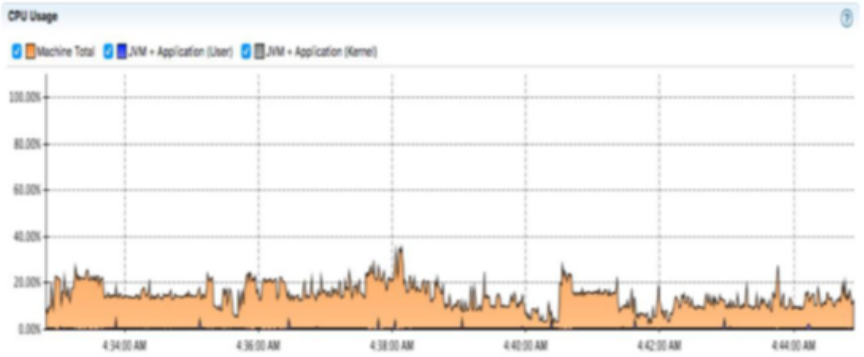

FIGURE 12 CPU Usage of Refactored System

TABLE 6 Transaction Processing Error Rate

Version		Error Rate(%))			
		Iteration1	**Iteration2**	**Iteration3**	**Iteration4**
Existing	SC1	41.05	43.00	38.86	40.97
	SC2	8.45	7.93	6.57	7.65
	SC3	9.86	10.18	6.86	8.97
	SC4	26.91	27.02	26.86	26.83
Refactored	SC1	24.40	37.53	28.48	30.14
	SC2	6.55	8.57	5.69	6.94
	SC3	7.37	8.29	7.23	7.63
	SC4	34.27	16.54	22.53	24.45

TABLE 7 Transaction Processing Error Rate Conformance

Version	**SC1**	**SC2**	**SC3**	**SC4**	**Total**
Existing	40.97	7.65	8.97	26.93	84.52
Refactored	30.14	6.94	7.63	24.45	69.16
Transaction Processing Error Rate Conformance	0.74	0.91	0.85	0.91	0.82

6.2.6 | Transaction Capacity

The transaction processing error rate is the percentage of transactions the CPU can not completely process for a given task. This situation usually happens because increasing requests attack the server over its processing capacity. In our experiments, the number of 350 users is identified as the maximum number of requests the CPU can handle. When the user is increased to more than the maximum number, some requests fail to be processed and generate an error.

Table 6 shows that transaction error rates of the existing system are higher than those of the refactored system. This situation is depicted in Figure 13 respectively. The experiment results show that the average transaction processing error rate for the existing system is 21.13%, while for the refactored system is 17.29%. Moreover, Table 7 shows that all of the transaction processing error rate conformance values are less than 1, indicating that the refactored system is better than the existing system in terms of its processing capacity.

Those results prove that adding an extra facade layer causes better thread management between layers. This happens because message calling between layers can be reduced, resulting in fewer requests that fail to be processed. This section concludes that applying design patterns may increase the quality of capacity measures.

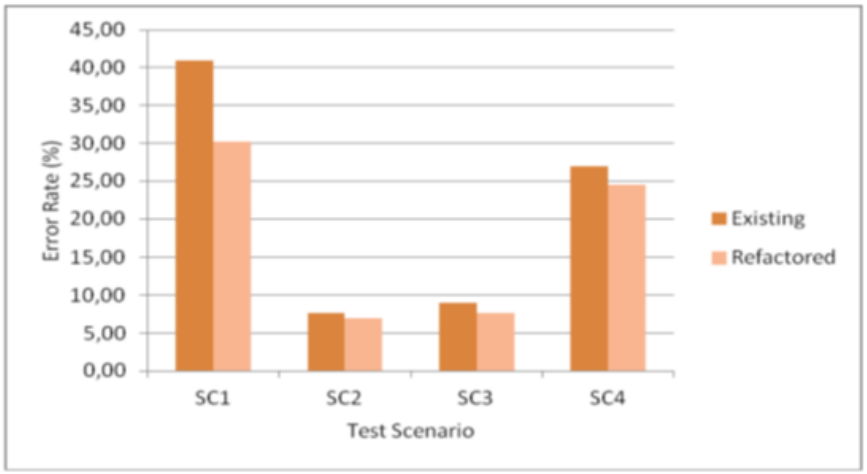

FIGURE 13 Chart of transaction Processing Error Rate

7 | CONCLUSION

This research has presented a methodology for conducting experiments to demonstrate the impact of applying design patterns on application complexity and performance. The SIA system, specifically the Assessment module, is used as a case study in the experiments by adding a facade layer to the current module version.

The experimental results show that applying the facade layer decreases program complexity despite increasing the number of artifacts. In addition, the transaction processing error rate decreases, indicating that the refactored system works better in completing requests. However, the extra layer of indirection also increases response time and resource utilization.

Hence, the overall conclusion related to software quality is that the design pattern may decrease the quality of time behavior and resource utilization while increasing the quality of capacity measures and complexity to a signisignificantlysults can be used as a starting point for elaborating the impact of applying design patterns to other quality aspects. In addition, experiments should also be carried out to apply other types of design patterns as they may produce different values.

ACKNOWLEDGMENT

We would like to acknowledge Department of Informatics and Software Engineering Laboratory, Institut Teknologi Sepuluh Nopember, for facilitating and funding this research.

CREDIT

Siti Rochimah: Conceptualization, Methodology, Validation, Resources, Data Curation, Writing - Original Draft, and Visualization. **Rizky Januar Akbar:** Methodology, Validation, Writing - Review & Editing, and Supervision. **Kholed Langsari:** Software, Investigation, and Formal analysis.

References

1. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-oriented Software. Boston, USA: Addison-Wesley Longman Publishing Co., Inc; 1995. https://www.javier8a.com/itc/bd1/articulo.pdf.

2. Demeyer S, Ducasse S, Nierstrasz O. Object-Oriented Reengineering Patterns. San Fransisco, USA: Morgan Kaufmann Publishers Inc; 2002. https://www.sciencedirect.com/book/9781558606395/object-oriented-reengineering-patterns?via=

ihub=.

3. Ulrich W, Gotelli NJ. Pattern detection in null model analysis. Oikos 2013;122(1):2–18. https://nsojournals.onlinelibrary.wiley.com/doi/10.1111/j.1600-0706.2012.20325.x.

4. Ali N, Elish MO. A Comparative Literature Survey of Design Patterns Impact on Software Quality. In: International Conference on Information Science and Applications Pattaya, Thailand: June; 2013. p. 1–7. https://ieeexplore.ieee.org/document/6579460.

5. Suryn W. Software Quality Engineering: A Practitioner's Approach. January; 2014. https://onlinelibrary.wiley.com/doi/book/10.1002/9781118830208.

6. Khomh F, Gueheneuce Y. An Empirical Study of Design Patterns and Software Quality. In: 12th European Conference on Software Maintenance and Reengineering Athens, Greece: IEEE; 2008. p. 274–278. https://www.computer.org/csdl/proceedings/csmr/2008/12OmNwCJOWz.

7. Rochimah S, Rahmani HI, Yuhana UL. Usability Characteristic Evaluation on Administration Module of Academic Information System using ISO/IEC 9126 Quality Model. In: International Seminar on Intelligent Technology and Its Applications (ISITIA) Surabaya, Indonesia: IEEE; 2015. p. 363–366. https://ieeexplore.ieee.org/document/7220007.

8. Yuhana UL, Saptarini I, Rochimah S. Portability characteristic evaluation Academic information System assessment module using AIS Quality Instrument. In: 2nd International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE) IEEE; 2015. p. 133–137. https://ieeexplore.ieee.org/document/7437785.

9. IEEE. ISO/IEC/IEEE Standard for Systems and Software Engineering - Software Life Cycle Processes. ISO/IEC/IEEE 12207-2:2020(E) 2008;p. 138. https://www.mendeley.com/search/?page=1&query=ISO%2FIEC%2FIEEE%20Standard%20for%20Systems%20and%20Software%20Engineering%20-%20Software%20Life%20Cycle%20Processes&sortBy=relevance.

10. Perdomo W, Zapata CW. Software Quality Measures and Their Relationship with the States of the Software System Alpha. Ingeniare 2021;29(2):346–363. https://www.scielo.cl/scielo.php?script=sci_arttext&pid=S0718-33052021000200346&lng=en&nrm=iso&tlng=en.

11. Rochimah S, Hantriono AI, Akbar RJ, Baskara AR. Application of design patterns and quality measurement on academic information systems. 4th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE) 2017;https://ieeexplore.ieee.org/document/8257669.

12. Gautama IMB, Rochimah S, Akbar RJ. Assessing the Impact of Enterprise Software Design Patterns on Maintainability: A Case Study. In: 1st International Conference on Cybernetics and Intelligent System (ICORIS) Denpasar, Indonesia: IEEE; 2019. p. 128–132. https://ieeexplore.ieee.org/document/8874885.

13. Rochimah S, Nuswantara PG, Akbar RJ. Assessing the Impact of Enterprise Software Design Patterns on Maintainability: A Case Study. In: Electrical Power, Electronics, Communications, Controls and Informatics Seminar (EECCIS) Batu, Indonesia: IEEE; 2018. p. 326–331. https://ieeexplore.ieee.org/document/8692876.

14. Nazar N, Aleti A, Zheng Y. Feature-based software design pattern detection. Journal of Systems and Software 2022;185:111179. https://www.sciencedirect.com/science/article/pii/S0164121221002624.

15. Henderson, Sellers. Object-Oriented Metrics: measures of Complexity. Hoboken, New Jersey, U.S.: Prentice-Hall; 1996. https://www.mendeley.com/search/?page=1&query=Object-oriented%20Metrics%3A%20Measures%20of%20Complexity&sortBy=relevance.

16. Saca MA. Refactoring improving the design of existing code. In: IEEE 37th Central America and Panama Convention IEEE; 2017. p. 1–3. https://ieeexplore.ieee.org/document/8278488.

17. Suryanarayana G, Samarthyam G, Sharma T. Refactoring for Software Design Smells: Managing Technical Debt. 1st ed. Burlington, Massachusetts: Morgan Kaufmann; 2014. https://www.sciencedirect.com/book/9780128013977/

refactoring-for-software-design-smells?via=ihub=.

How to cite this article: Siti Rochimah, Rizky Januar Akbar, Kholed Langsari (2024), Investigating Design Patterns Impact on Application Performance and Complexity, *IPTEK The Journal of Technology and Science*, *35(1):1-17*.