THESIS - KI142502

THE IMPACT OF DESIGN PATTERNS IN REFACTORING TECHNIQUE TO MEASURE PERFORMANCE EFFICIENCY

Kholed Langsari
NRP : 5115201701

SUPERVISORS
Dr. Ir. Siti Rochimah, M.T.
Rizky Januar Akbar, S.Kom., M.Eng.

MAGISTER PROGRAM
SOFTWARE ENGINEERING
DEPARTMENT OF INFORMATICS
FACULTY OF INFORMATION TECHOLOGY
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA
2017

Tesis disusun untuk memenuhi salah satu syarat memperoleh gelar
Magister Komputer (M.Kom.)
di
Institut Teknologi Sepuluh Nopember Surabaya

oleh:
Kholed Langsari
Nrp. 5115201701

Dengan judul :
The Impact of Design Pattern in Refactoring Technique to Measure Performance Efficiency

Tanggal Ujian : 17-7-2017
Periode Wisuda : 2016 Genap

isetujui oleh:

[illegible]

r. Ir. Siti Rochimah, M.T
IP. 196810021994032001 (Pembimbing 1)

[illegible]

izky Januar Akbar, S.Kom, M.Eng
IP. 198701032014041001 (Pembimbing 2)

[illegible]

aniel Oranova Siahaan, S.Kom, M.Sc, PD.Eng.
IP. 197411232006041001 (Penguji 1)

[illegible]

ajar Baskoro, S.Kom, M.T
IP. 197404031999031002 (Penguji 2)

[illegible]

dhatus Sholichah, S.Kom, M.Sc
IP. 198508262015042002 (Penguji 3)

KEMENTERIAN RISET TEKNOLOGI DAN PENDIDIKAN TINGGI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
TEKNOLOGI INFORMASI

Dekan Fakultas Teknologi Informasi,

[illegible]

Dr. Agus Zainal Arifin, S.Kom, M.Kom.
NIP. 197208091995121001

(This page intentionally left blank)

THE IMPACT OF DESIGN PATTERNS IN REFACTORING TECHNIQUE TO MEASURE PERFORMANCE EFFICIENCY

Name	: Kholed Langsari
Student Identity Number	: 5115201701
Supervisor I	: Dr. Ir. Siti Rochimah, MT.
Supervisor II	: Rizky Januar Akbar, S.Kom., M.Eng.

ABSTRACT

Designing and developing software application has never been an easy task. The process is often time consuming and requires interaction between several different aspects. It will be harder in re-engineering the legacy system through refactoring technique, especially when consider to achieve software standard quality. Performance is one of the essential a quality attribute of software quality.

Many studies in the literature have premise that design patterns improve the quality of object-oriented software systems but some studies suggest that the use of design patterns do not always result in appropriate designs. There are remaining question issues on negative or positive impacts of pattern refactoring in performance. In practice, refactoring in any part or structure of the system may take effect to another related part or structure. Effect of the process using refactoring technique and design patterns may improve software quality by making it more performable efficiency. Considerable research has been devoted in re-designing the system to improve software quality as maintainability and reliability. Less attention has been paid in measuring impact of performance efficiency quality factor.

The main idea of this thesis is to investigate the impact, demonstrate how design patterns can be used to refactor the legacy software application in term of performance efficiency. It is therefore beneficial to investigate whether design patterns may influence performance of applications. In the thesis, an enterprise project named SIA (Sistem Informasi Akademik) is designed by applying Java EE platform. Some issues related to design patterns are addressed. The selection of design pattern is based on the application context issue.

There are three kind of parameters measure, time behavior, resource utilization and capacity measures that based on standard guideline. We use tools support in experimentation as Apache JMeter and Java Mission Control. These tools provide convenient and generate appropriate result of performance measurement. The experiment results shown that the comparison between the legacy and refactored system that implemented design pattern indicates influence on application quality, especially on performance efficiency.

***Key Words**: Impact, Design Patterns, Refactoring, Software Quality, Performance Efficiency*

(This page intentionally left blank)

DAMPAK POLA DESAIN DALAM TEKNIK REFAKTOR UNTUK MENGUKUR EFISIENSI KINERJA

Nama Mahasiswa	: Kholed Langsari
NRP	: 5115201701
Pembimbing I	: Dr. Ir. Siti Rochimah, MT.
Pembimbing II	: Rizky Januar Akbar, S.Kom., M.Eng.

ABSTRAK

Merancang dan mengembangkan aplikasi perangkat lunak bukan merupakan pekerjaan yang mudah karena membutuhkan waktu dan interaksi antara beberapa aspek. Proses desain pada rekayasa ulang akan lebih sulit meskipun melalui teknik refactoring, terutama untuk mencapai standar kualitas perangkat lunak. Kinerja merupakan salah satu atribut terpenting kualitas perangkat lunak.

Banyak penelitian menjelaskan pola desain memperbaiki kualitas sistem perangkat lunak berorientasi objek, namun beberapa penelitian juga menunjukkan bahwa penggunaan pola desain tidak selalu menghasilkan desain yang sesuai. Masih ada pertanyaan tentang dampak negatif atau positif dari kinerja pola refactoring. Pada praktiknya, melakukan refactoring pada suatu bagian atau struktur sistem akan berpengaruh pada bagian atau struktur lain yang terkait. Penggunaan teknik refactoring dan pola desain dapat meningkatkan kualitas perangkat lunak dengan kinerja lebih efisien. Sudah banyak penelitian yang berfokus untuk merancang ulang sistem untuk meningkatkan kualitas perangkat lunak sebagai kemampuan rawatan dan keandalan. Tetapi masih kurang penelitian perhatian dalam mengukur dampak faktor kualitas efisiensi kinerja.

Tujuan utama dalam tesis ini adalah untuk mengetahui dampaknya, menunjukkan bagaimana pola desain dapat digunakan untuk refactor aplikasi perangkat lunak terdahulu dalam hal efisiensi kinerja. Oleh karena itu, akan bermanfaat untuk menyelidiki apakah pola desain dapat mempengaruhi kinerja aplikasi. Dalam tesis ini, sebuah proyek perusahaan bernama SIA (Sistem Informasi Akademik) dirancang dengan menerapkan platform Java EE. Beberapa masalah yang terkait dengan pola desain diketahui. Pemilihan pola desain berdasarkan pada isu konteks aplikasi.

Tiga jenis ukuran parameter dipakai untuk penilitian ini, perilaku waktu, pemanfaatan sumber daya dan ukuran kapasitas yang berdasarkan pada pedoman standar. Kami menggunakan Apache JMeter dan Java Mission Control sebagai alat bantu dalam eksperimen. Hasil percobaan menunjukkan bahwa perbandingan antara sistem terdahulu dengan penelitian ini yang menerapkan pola desain menunjukkan bahwa hasilnya berpengaruh terhadap kualitas aplikasi terutama pada efisiensi kinerja.

***Key Words**: Impact, Design Patterns, Refactoring, Software Quality, Performance Efficiency*

(This page intentionally left blank)

ACKNOWLEDGEMENTS

The preparation of this thesis on the topic of “The Impact of Design Patterns in Refactoring Technique to Measure Performance Efficiency” would not be completed without the contribution and support of both individual and institutions, and absolutely this thesis title would not be chosen without the desired of Allah SWT.

I am grateful to Allah SWT, all praise [due] to him, the Entirely Merciful and the Especially Merciful and only one who helps me to complete this thesis.

I would like to thank to my big warm families especially for my parents who stood by me regardless of many obstacles.

I would like to thank to DIKTI (Pendidikan Tinggi Republik Indonesia) and Fatoni University who support in this education, especially in financials.

I would like to thank to my supervisors, Dr. Ir. Siti Rochimah, MT., and Rizky Januar Akbar, S.Kom., M.Eng., who suggest to do and advise me to complete this thesis since started till finished.

I would like to thank to the members of my committees: Daniel Oranova Siahaan, S.Kom, M.Sc, PD.Eng., Laurent Balmelli, Fajar Baskoro, S.Kom, M.T, and Adhatus Sholichah, S.Kom, M.Sc for spending their time to evaluate my work.

I would like to thank to the members of the software engineering group (LBE) for the working environment I have had.

I would like to thank all of lecturers, and stuffs of Institut Teknologi Sepuluh Nopember, especially for Department of Informatics Engineering, who give a good help and advice for me.

I would like to thank to all brothers and sisters that stayed in Indonesia. They have helped me not to feel homesick. I would like to thank to all friends I have in the Indonesia who have always together against obstacles.

(This page intentionally left blank)

TABLE OF CONTENTS

(This page intentionally left blank)

TABLE OF FIGURES

TABLE OF TABLES

(This page intentionally left blank)

CHAPTER 1
INTRODUCTION

1.1 Background

Designing and developing software application has never been an easy task. The process is often time consuming and requires interaction between several different aspects. The enterprise software developers are making efforts to develop the enterprise software application that is not only satisfy the business needs but also achieve the high quality within a short development process. Several tools, notations, principles, and methods have been proposed. The Object-Oriented approach have been introduced to guide the development process and offering how to design OO systems (Larman, 2004). The Unified Modelling Language (UML) (Priestley, 2003) is commonly used to model the design, vocab like class and object denote commonly accepted concepts.

Even the most complex systems are built by using smaller parts. Such parts may in built using even smaller parts and need to communicate to function as a whole. The OO approach attempts to manage the system complexity by abstracting out knowledge and encapsulating it within interacting objects. A part can be viewed as a single object or a collection of interacting objects delivering a specific functionality. If we view a part as a design problem to be solved, regardless of the approach chosen, it is likely that others have already solved a similar problem in a satisfactory manner. If we can utilize this knowledge, the quality of the system may be improved.

There is an approach use to identify reoccurring design problems and their well-proven solutions, that is software design patterns. The concept of design patterns (Gamma, Helm, Johnson, & Vlissides, 1995) have been has been present in software engineering for a relatively long time. A design pattern is an abstraction of practical experience and empirical knowledge description of the problem it addresses and a solution to it. Design patterns function in software engineering along with other pattern categories, for instance, reengineering patterns (Demeyer, Ducasse, & Nierstrasz, 2002) or analysis patterns (Fowler, 1997). Design patterns can facilitate the entire design and development process because they express ideas and solutions in high level language. The use of patterns may improve software quality by making it more reusable, maintainable, performable efficiency. Developers are increasingly more aware of how and when to use different kinds of patterns.

Performance is one of the important and essential a quality attribute of software quality (Ali & Elish, 2013). Performance of an application is particularly important for a customer ordering a piece of software. Client of software usually regard performance as an important standard to decide whether the software is good to use. Usually, it is not essential for the client to know what kinds of design decisions were made. However, it is far more important to know how the software performs, whether its services are reliable and available for end-users as expected. The performance of software reflects the efficiency of software, because the software which makes proper use of resources is usually responding fast. Performance is an important internal and external quality attribute, which can be measured as throughput and resource utilization (Suryn, 2014). However, performance is only one of many parameters of an application that determine the quality of the final product. Performance-related aspects can be categorized and characterized by time behavior, resources utilization and capacity compliance (Suryn, 2014).

The main idea of this thesis is to investigate the impact and demonstrate how design patterns can be used to refactor the legacy software application in order to achieve the quality requirement, performance efficiency. In this thesis, an enterprise project named SIA (Sistem Informasi Akademik) in the field of education service is designed by applying Java EE platform, architecture and patterns. Some issues related to design patterns and Java EE are addressed. This thesis is conducted within the context of the Systems and Software Engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Measurement of system and software product quality (ISO/IEC, 2011a, 2011b). The SQuaRE international standard aims to defines quality measures for 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” design patterns through refactoring process to see the impact result in term of performance efficiency.

1.2 Problem Statement

Performance is non-functional requirement that important factor to consider in enterprise system in order to achieving high quality of application. In critical software system, performance become functional requirement and high priority consideration quality attribute as in banking system.

A design pattern is generally thought of as a reusable solution to a commonly occurring design problem in object-oriented software. The Gang of Four (GOF), define patterns are cataloged as: "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." (Gamma et al., 1995). Design patterns must be applied with caution. They provide solutions at a certain level of granularity, often the class level in a standard OO language. The problems are often centered around how to encapsulate variability in a design. Most of the catalogued patterns target as a whole the properties that flexible software. Patterns when implemented, often comes with the cost of an extra layer of indirection, creating the way for increased abstraction in the program. In the other way, there are also design patterns that can reduce object calls and layers as Singleton. It is created single handle object and call through it, it directly improves performance of application because of it reduce traditional OO design in creating extra layer and provide flexible of design.

Applying several design patterns might create several layers of indirection. This may have a positive or a negative impact on performance. Design pattern provide discipline in create or refactor to a better software structure but they cannot offer any guarantees in performance of the software quality. The true benefit is only realized if a given collection of design patterns is used on a regular basis in a specific domain and context.

As performance is an important factor in software quality. A survey of design patterns impact on software quality (Ali & Elish, 2013) (Khomh & Gueheneuce, 2008), have shown that performance is one of the quality attribute that affect quality of software system. But the number of studies and the coverage of the addressed patterns are not sufficient to draw a conclusion regarding their impact on performance. Many studies in the literature have for premise that design patterns improve the quality of object-oriented software systems, some studies suggest that the use of design patterns do not always result in appropriate designs (Khomh & Gueheneuce, 2008). There is accumulate and create remain question issues as "do refactoring and design patterns really impact performance improvement?" and "does it impacts negative or positive on performance?"

Below we present an overview of the problems addressed by this thesis:

1. **Explosion of Impacts in Refactoring Process using Design Patterns.** In practice, refactoring in any part or structure of the system may take effect to another related part or structure. Effect of the process through the use of refactoring technique and design patterns may improve software quality by making it more performable efficiency.
2. **Measurement Impact of Performance Efficiency when Implement Refactoring Technique using Design Patterns.** Considerable research has been devoted in re-designing the system to improve software quality as maintainability and reliability. Less attention has been paid in measuring impact of performance efficiency quality factor in refactoring and design pattern applied.

1.3 Problem Limitations

Thesis problem limitations list as below:

1. The refactoring process of this thesis is focused on design smell and code smell level of the software application. This thesis does not include database and architecture level of the software.
2. Implementation of design patterns is based on ones described by "Gang of Four" design patterns.
3. Design and model notation for design patterns are specified and modelled using UML standard.
4. The main case study is SIA, it is developed using Java (Enterprise Edition - EE) programming language with Spring MVC framework,

Hibernate and other support technology as JavaScript, JSON and jQuery. The database management system used is PostgreSQL.

1.4 Research Questions and Objectives

The objectives of this thesis are to investigate how refactoring process and design patterns can be used to re-design and refactor the legacy software application with considering performance efficiency. Answering these following questions will give a better understanding of the problems domain and the deficiencies of the current solution.

1. RQ 1: What are the current design patterns approaches, refactoring technique in software design and performance efficiency measurement?
2. RQ 2: How to analyze and refactor the existing system with design pattern perspective which respect to performance efficiency quality attribute?
3. RQ 3: How to measure performance efficiency of the system using standard performance efficiency measures?
4. RQ 4: How to evaluate performance efficiency of the system?

1.5 Significance of Study

This thesis study aims to identify some significant.

1. First, this study will discuss why performance efficiency are important in software design and software implementation phases. Especially when applying together with refactoring and design patterns.
2. Second, the thesis will summarize current approaches and mechanisms of the design patterns, refactoring technique, and performance measurement.
3. Finally, some challenge, benefit and impact of implementing design patterns through refactoring will also be discussed.

1.6 Contribution

The contribution of this thesis is to present a scientific evidence of the impact of design patterns through refactoring process to measure performance efficiency in particular environments. The case study is Academic Information System and the design patterns are implemented and defined based on “Gang of Four” design patterns.

1.7 Outline of the Thesis

Outline of the thesis present thesis with chapters and relations among them. The thesis consists of the following chapters:

CHAPTER 1 : INTRODUCTION

This chapter, start with introduction of the study, describe the problem addressed in this thesis, research questions, and objectives, problems limitations, significances of study, together with contribution and an outline of the thesis.

CHAPTER 2 : THEORY AND LITERATURE REVIEW

This chapter presents the review of the literature on concepts and techniques from the areas of software design and Object-Oriented approach, Patterns and Design Patterns, Refactoring, Software Quality and its characteristics and case study. And provide background information on these areas and introduce a set of definitions used throughout the study.

CHAPTER 3 : RESEARCH METHODOLOGY

This chapter describe research methodology and approach that going to implement.

CHAPTER 4 : RESULT AND DISCUSSION

This chapter presents the approach and implementation result from experiment phase and do the analyze to validate the output.

CHAPTER 5 : CONCLUSIONS

This chapter gives conclusions and evaluation of the contributions in this thesis, and describes directions for future work.

CHAPTER 2
THEORY AND LITERATURE REVIEW

In this work, we utilize concept and techniques from the areas of software design, object-oriented, pattern and design patterns, software quality, refactoring, and our case study. In this chapter, we provide background information on these areas and introduce a set of definitions used throughout the thesis.

2.1 Software Design

Software engineering is an engineering discipline for professional and systematic software development rather than individual programming that is concerned with all aspects of software production (Pressman, 2010). It includes aspects such as specification, development, validation, and evolution. The development is concerned of the designing and implementing the software. This section gives a more though definition of design and software design. We discuss the concept, principle, method and tool related.

2.1.1 Design Principles

Design principles provide guidance to designers in creating effective and high-quality software solutions.

Design is defined as both process of defining the architecture, component, interface, and other characteristics of a system or component and the result of that process (Pressman, 2010). In standard list of software life cycle process as ISO/IEC/IEEE Std. 12207-2008 ("ISO/IEC/IEEE Standard for Systems and Software Engineering - Software Life Cycle Processes," 2008), define software design consist of two activities, that are software architecture design and software detailed design

In general view, software design can be viewed as a form of problem solving. Software design is a process that is usually made by using the results of requirement analysis. Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. The goal of design is to produce a model or representation that exhibits firmness, commodity and delight. The purpose of the software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used (Sommerville, 2010). The design process can include multiple iterations before the final design is achieved. In this thesis, we focus on software detail design.

2.1.2 Object-Oriented

The Object-Oriented (OO) approach to software design attempts to manage the complexity inherent in real world problems by abstracting out knowledge and encapsulating it within objects. A complete discussion of this topic is found in the books of (Booch, 2004) (Priestley, 2003).

The basic element in an object-oriented system is an object. The focus of object–oriented is on decomposing the problem into objects. Object-Oriented development is a method of implementation in which, programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. In such programs, classes are viewed as static whereas objects typically have a much more dynamic nature.

Object orientation is a technique for software system. These techniques lead to design architectures based on objects that are manipulated in every system. OO design systems are better modeled domain systems than similar systems created by structure systems. It offers a number of concepts, which are well suited for this purpose. By understanding these object-oriented concepts; designers will learn how to apply those concepts to all stages of the software development life cycle. In the following subsections, we will introduce the basic concepts within object-oriented environment.

Object (Weisfeld, 2013) (Booch, 2004). An object is a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand. An object entity with some state, some behavior, and an identity. The structure and behavior of similar objects are grouped in their common class. The terms instance and object are interchangeable. All information in object-oriented system is stored within its objects. The aim of object-oriented approach is to decompose the problem into cooperating objects. The property new in object–oriented is to use objects as the important abstractions and to decompose the problem into object rather than using the traditional algorithmic decomposition.

Class. A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their information structure (Weisfeld, 2013). An actual understanding of a class that consists of data and the operations related with that data are important. It is an item that a user can manipulate as a single unit to perform a task. A class represents a set of objects that share a common structure and a common behavior. In an OO environment, a class is a specification of instance variables, methods, and inheritance for objects. Once a class is defined, any number of objects can be created which belong to that class i.e. class is everything about objects where objects are individual instance of a class.

Inheritance. Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship" (Weisfeld, 2013). It is the process by which objects of one class acquire the properties of the objects of another class. In OO design, the concept of inheritance supports the idea of reusability. By inheritance, it is possible to add new features to an existing class without modifying the previous class, so this is the way to derive a new class from an existing one. The new class is called a subclass or a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations (Gamma et al., 1995).

Polymorphism. Program entities should be permitted to refer to objects of more than one class, and operations should be permitted to have different realizations in different classes (McConnell, 2004). Polymorphism means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class (Weisfeld, 2013). Polymorphism means the ability to take more than one form. Through polymorphism, it is possible to hide many implementations behind the same interface. Polymorphism is a concept in type theory, according to which a name may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways (Booch, 2004). Polymorphism plays an important role in allowing objects to have different internal structures but share the same external interface. This means that a general class of operations can be accessed in the same manner, even though specific actions associated with each operation may differ.

Encapsulation. Encapsulation is a mechanism to realize data abstraction and information hiding. Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics (Booch, 2004). It hides detailed internal specification of an object, and publishes only its external interfaces. Thus, users of an object only need to hold on to these interfaces. By encapsulation, the internal data and methods of an object can be changed without changing the way of using the object. By hiding a representation and implementation in an object, more reusable specialized classes can be created. The representation cannot be accessed and is not visible directly from the object. Operations are the only way to access and in modify an object's representation (Gamma et al., 1995). Encapsulation is the most remarkable feature of a class. The data is not accessible to the outside world, only those functions, which are wrapped in the class, can access it. Encapsulation is a principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision (Weisfeld, 2013). In object-oriented approach, by using encapsulation a designer makes design easier, less annoying, more sustainable, and more efficiently workable.

Aggregation. An aggregate is a union of several objects, and the union as such is often represented by an object its own (Weisfeld, 2013). Aggregation is a type of relationship between objects. Objects are organized into an aggregation structure that shows how one object is composed of many other objects. In aggregation, host object acts as a relationship between the outside world and an inner object. (Priestley, 2003).

The most common used of model notation for OO approach is Unified Modeling Language (UML). UML is the standard modelling language for object-oriented systems. The UML is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system (Fowler, 2003). Structural and behavioral aspects of systems may be captured by a series of different kinds of models such as Class, Object, Use case, Sequence diagrams and so on. Design patterns are often described with UML in various pattern books (Larman, 2004) (Fowler, 1997) (Gamma et al., 1995) (Hunt, 2003).

Our case study SIA implemented object-oriented with Java Enterprise Edition (Java EE) as it main approach and tool in development. SIA have been designing, modeling, and documenting in the standard of UML. In this work, we use UML as our notation and description standard in several phases of methodology and implementation.

2.2 Patterns

Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns: These problem-solution pairs tend to fall into families of similar problems and solutions with each family exhibiting a pattern in both the problems and the solutions (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996). The architect Christopher Alexander defines the term pattern as follows:

1. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
2. As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
3. As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.
4. The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing. and when we must create it. It is both a process and a thing: both a description of a thing which is alive, and a description of the process which will generate that thing.

Software patterns first became popular with the wide acceptance of the book Design Patterns: Elements of Reusable Object-Oriented Software (Gamma et al., 1995), Pattern-Oriented Software Architecture: A System of Patterns (also called the POSA book, consist of 5 series) (Buschmann et al., 1996) and books Pattern Languages of Program Design and Pattern Languages of Program Design (Aguiar & David, 2009) (consist of three series).

Experts in software engineering know the patterns gain from practical experience and follow them in developing applications with specific properties. They use them to solve design problems both effectively and elegantly. The authors of Patterns of Software Architecture (Buschmann et al., 1996) define these three types of patterns as follows: Architectural Patterns, Design Patterns and Idioms. Here we specific discuss on Design Patterns.

2.2.1 Design Patterns

Design patterns are defined by Gamma et. al. (Gamma et al., 1995) as simple and elegant solutions to a recurring specific problems arising when designing object-oriented software design.

A pattern describes a problem that frequently occurs and proposes a possible solution in terms of the organization of classes and objects that are generally recognized like a good solution to solve the problem. Design patterns are also reusable, meaning they are used for a variety of situations in many different architectures. They realize a generic solution for a set of functional requirements. More importantly, they are simple and elegant, which allows developers to easily understand them and extend them without modifying existing classes and increasing code complexity.

A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structures of communicating components that solve a general design problem within a particular context (Buschmann et al., 1996).

2.2.2 Classification of Design Patterns

There are four essential parts of a design pattern: The pattern name, the problem description, the solution, and the consequences of the application of a certain pattern. These parts are described below.

The pattern name. conveys the essence of the pattern. The name is used as part of the design vocabulary to describe solutions to certain problems. The name of the pattern makes it easier to talk about a design, to document a design, and even to think about a design. Thus, it can be used as an abstraction at a higher level.

The problem description motivates the general problem solved by the design pattern. This defines when to apply the pattern. This part of the pattern describes a general situation which has to be solved. Sometimes a concrete example is used to do so. The problem description might include a list of conditions which must hold before the application of the pattern makes sense.

The solution describes the elements of the pattern together with their interaction, their responsibility, and their relationships. The description of the solution is not dependent on a concrete problem and there is no implementation given which is used to always solve the problem: The exact layout of the objects involved in a pattern depends on the actual problem to be solved. The design pattern only gives a general arrangement of objects and classes involved in the solution of a general problem. This solution has to be tailored and adapted to the problem actually solved.

The consequences section lists the trade–offs made when applying a design pattern. This is used basically to decide whether the approach of the design pattern is feasible to solve a certain problem: There may be other design patterns solving a problem which is quite similar to the problem solved by the design pattern at hand but with different trade–offs. In addition, the costs involved in the application of a design pattern may be too high to solve a certain problem such that a different solution has to be sought. The consequences of design patterns help to evaluate the impact on a system’s extensibility, flexibility, and portability but also list the costs and limitations.

The design patterns are classified by two criteria (see Table 2.1). The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The second criterion, called scope, specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static-fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So, the only patterns labeled class patterns are those that focus on class relationships.

Table 2.1 Design pattern catalog

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

2.2.3 Hierarchical model–view–controller (HMVC)

Model-View-Controller (MVC) is an architectural pattern that describes a way to structure of an application and the responsibilities and interactions for each part in that structure by split user interface interaction into three distinct roles (Buschmann et al., 1996). MVC divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change propagation mechanism ensures consistency between the user interface and the model.

The Hierarchical-Model-View-Controller (HMVC) is a software architectural pattern. HMVC is a direct extension to the MVC pattern that manages to solve many of the scalability issues. HMVC is a collection of traditional MVC triads operating as one application. Each triad is completely independent and can execute without the presence of any other. All requests made to triads must use the controller interface, never loading models or libraries outside of their own domain. The triads physical location within the hosting environment is not important, as long as it is accessible from all other parts of the system. The distinct features of HMVC encourages the reuse of existing code, simplifies testing of disparate parts of the system and ensures that the application is easily to enhanced or extended.

To successfully design applications that implement the HMVC pattern, it is critical that all the application features are split down into systems. Each system is one MVC triad within the larger HMVC application, independently managing presentation and persistent storage methods. Presently few frameworks are available that support HMVC without additional extensions, or use inefficient Front Controllers and dispatching.

2.2.4 Façade Pattern

Façade is a structural purpose type and object scope design patterns of Gang of Four (GoF). Façade provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use (Gamma et al., 1995). Basically, this is saying that we need to interact with a system that is easier than the current method, or we need to use the system in a particular way. We can build such a method of interaction because we only need to use a subset of the system in question.

Facade simplifies complex code, making it easier to use poorly designed, over-complex subsystems. It meant to be wrappers around complex functionality, their primary goal is hiding complexity of an underlying system.

Facade provides a simple interface to a complex subsystem, a single interface through which all the classes in a complex subsystem (sub-system classes: classes that comprise one or more complex subsystems) are manipulated.

Facade allows to treat a complex subsystem as if it were a single course-grained object with a simple easy-to-use interface. Figure 4.11 described class diagram overview of Façade design pattern.

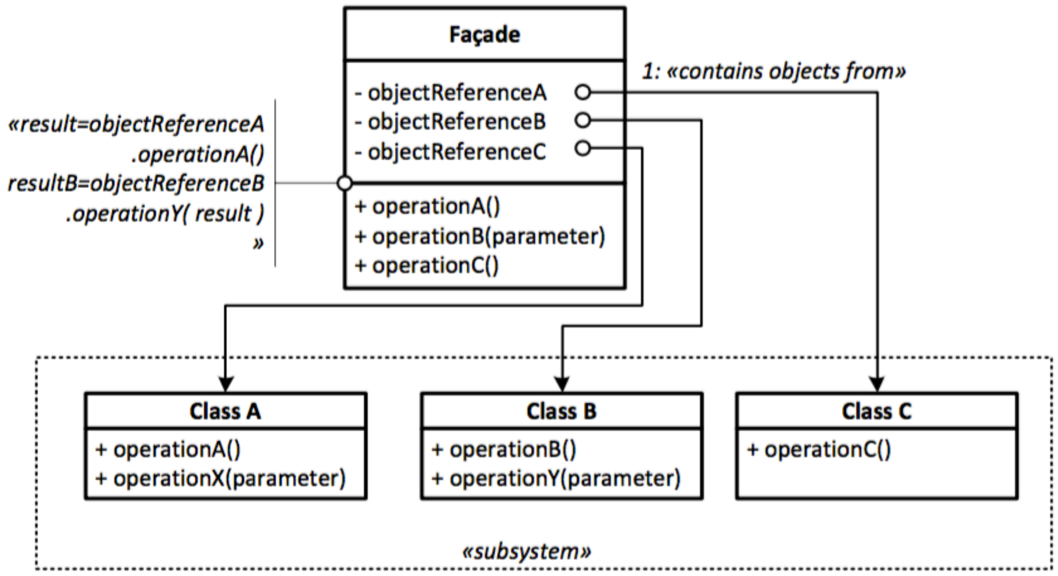


Figure 2.1 Class diagram of the Façade pattern

The advantages of using Façade are reduced coupling relationships between subsystems, improving maintenance and flexibility but programmers still possible to ignore the Facade and use subsystem classes directly.

2.3 Software Quality

Quality is a fundamental property of software systems and generally refers to the degree to which a software system lives up to the expectation of satisfying its requirements (Suryn, 2014). Quality is often characterized in terms of attributes such as modifiability, durability, interoperability, portability, security, predictability, scalability, efficiency and so on. Some of these properties are software properties while others are system properties.

Despite a focus on software, Software quality refer to desirable characteristics of software products, to the extent to which a particular software product possesses those characteristics, and to processes, tools, and techniques used to achieve those characteristics (Society, Bourque, & Fairley, 2014). IEEE Std 1061 (the IEEE Standard for a Software Quality Metrics Methodology) provides a definition of software quality as "software quality is the degree to which software possesses a desired combination of attributes" (IEEE Computer Society, 2009).

2.3.2 Quality Attributes

Given that quality is the manifestation of the exhibited QAs, it makes sense for us to have a thorough understanding of which specific attributes are being considered. Fortunately, there are several existing frameworks that can assist in this regard. Such frameworks provide standard description and a useful checklist that can be used when gathering stakeholder requests or when reviewing requirements.

The international standard ISO/IEC 9126, Software engineering - Product quality (ISO/IEC, 2001), classifies software quality within a taxonomy of characteristics and sub-characteristics. The characteristics considered are functionality, reliability, usability, efficiency, maintainability, and portability. Each of these characteristics is further subdivided into sub-characteristics that themselves are subdivided into QAs that can be measured and verified.

1. **Functionality** considers a set of sub-characteristics that have a bearing on the function of the system in addressing the needs of stakeholders. The sub-characteristics considered are suitability, accuracy, interoperability, security, and functionality compliance.
2. **Reliability** considers a set of sub-characteristics that have a bearing on the ability of the software to maintain its level of performance under stated conditions for a stated period of time. The sub-characteristics considered are maturity, fault tolerance, recoverability, and reliability compliance.
3. **Usability** considers a set of sub-characteristics that have a bearing on the ease with which the software can be used by a known set of users. The sub-characteristics considered are understandability, learnability, operability, attractiveness, and usability compliance.

4. **Efficiency** considers a set of sub-characteristics that have a bearing on the relationship between the level of performance of the software and the amount of resources used under given conditions. The sub-characteristics considered are time behavior, resource utilization, and efficiency compliance.
5. **Maintainability** considers a set of sub-characteristics that have a bearing on the effort needed to make specified modifications. The sub-characteristics considered are analyzability, changeability, stability, testability, and maintainability compliance.
6. **Portability** considers a set of sub-characteristics that have a bearing on the potential for the software to be moved from one environment to another. The sub-characteristics considered are adaptability, installability, coexistence, replaceability, and portability compliance.

Another framework is the "FURPS+" classification (Grady, 1992), where the FURPS acronym stands for "functionality, usability, reliability, performance, and supportability" and the "+" represents any additional considerations that the system must accommodate. The sub-characteristics are taken from (Eeles & Cripps, 2010). In Performance software quality sub-characteristics can be composite as speed, Efficiency, Resource consumption, Throughput, and Response time (Grady, 1992)

ISO/IEC 2502n - Quality Measurement Division. The standards that form this division include a system/software product quality measurement reference model, definitions of quality measures, and practical guidance for their application. This division presents internal measures of software quality, external measures of software quality, quality in use measures and data quality measures. Quality measure elements forming foundations for the quality measures are defined and presented.

The International Standard, ISO/IEC 25023 – Measurement of system and software product quality is a part of 2502n Quality Measurement Division of SQuaRE series. The International Standard, ISO/IEC 25023 is defining quality measures for quantitatively measuring system and software product quality in terms of characteristics and sub-characteristics is defined, and intended to be used together with ISO/IEC 25010 - System and software quality models. The ISO/IEC

25010 categorizes software quality attributes into eight characteristics (functional suitability, reliability, performance efficiency, operability, security, compatibility, maintainability and transferability) as illustrated in Figure 2.1.

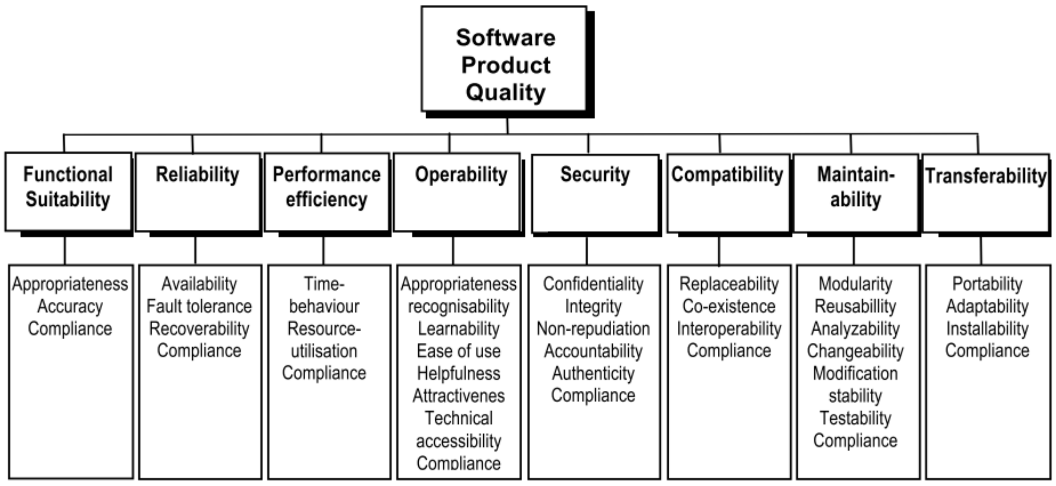

Software Product Quality
Functional Suitability
Reliability
Performance efficiency
Operability
Security
Compatibility
Maintain-ability
Transferability
Appropriateness Accuracy Compliance
Availability Fault tolerance Recoverability Compliance
Time-behaviour Resource-utilisation Compliance
Appropriateness recognisability Learnability Ease of use Helpfulness Attractivenes Technical accessibility Compliance
Confidentiality Integrity Non-repudiation Accountability Authenticity Compliance
Replaceability Co-existence Interoperability Compliance
Modularity Reusabillity Analyzability Changeability Modification stability Testability Compliance
Portability Adaptability Installability Compliance

Figure 2.2 Software Product Quality Model

We are focusing on performance efficiency characteristic and its sub-characteristics.

2.3.3 Performance Efficiency

There is an old saying that “what you do not measure you cannot control.” As a general definition, performance measures how effective is a software system with respect to time constraints and allocation of resources. Performance is a very important attribute of software. It used to be the main driving forces behind the development of software.

Performance is concerned with how well the software response when an event occurs (Rudzki, 2005). The software system events arrive in various patterns which can be characterized as periodic or stochastic. To evaluate whether a system is well performing, the time between the event and the response can firstly be measured, then compared with a previously determined time constrain.

Different resources used different terms and terminology of performance / efficiency / performance efficiency. Here we summarize important and often appear term and terminology related to performance from several resources:

Table 2.2 Summarize terms and terminology of performance

Terms	**Terminology**
Performance and **efficiency** ("IEEE Standard Glossary of Software Engineering Terminology," 1990)	The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage. And efficiency refer to the degree to which a system or component performs its designated functions with minimum consumption of resources. See also: execution efficiency; storage efficiency.
Performance (Grady, 1992)	Considers the degree to which the system provides a defined level of execution performance. This includes a consideration of speed, efficiency, resource consumption, throughput, and response time.
Efficiency (ISO/IEC, 2001)	Considers a set of sub-characteristics that have a bearing on the relationship between the level of performance of the software and the amount of resources used under given conditions. The sub-characteristics considered are time behavior, resource utilization, and efficiency compliance.
Efficiency (Suryn, 2014)	Optimum use of system resources during correct execution.
Performance Efficiency (ISO/IEC, 2011a, 2011b)	The degree to which the software product provides appropriate performance, relative to the amount of resources used, under stated conditions.

Performance Efficiency (ISO/IEC, 2011a) is the degree to which the software product provides appropriate performance, relative to the amount of resources used, under stated conditions. There are three main categories:

1. **Time behavior.** The degree to which the software product provides appropriate response and processing times and throughput rates when performing its function, under stated conditions.

2. **Resource utilization.** The degree to which the software product uses appropriate amounts and types of resources when the software performs its function under stated conditions.
3. **Performance efficiency compliance.** The degree to which the software product adheres to standards or conventions relating

In measuring the satisfaction of performance efficiency, we use performance efficiency measures and metrics defined in ISO/IEC 25023 – Measurement of system and software product quality (ISO/IEC, 2011b). Performance efficiency measures are used to assess the performance relative to the amount of resources used under stated conditions. Resources can include other software products, the software and hardware configuration of the system, and materials. The detail description and measurement function of each attribute characteristics and sub-characteristics as below:

1. **Time behavior measures.** Time behavior measures are used to assess the degree to which the response and processing times and throughput rates of a product or system when performing its functions meet requirements. There are five sub-characteristics of time behavior measures. The description of each sub-characteristics and measurement function detail in Table 2.3.

Table 2.3 Time behavior measures

ID / Name	**Description**	**Measurement function**
PTb-1-G, **Mean response time**	What is the mean time taken by the system to respond to a user action or system event?	$X = \sum_{i=1 \text{ to } n} (A_i) / n$ A_i = Time taken by the system to respond to action or event i n = Number of response events measured
PTb-2-G, **Response time conformance**	How well does the system response time meet the specified target?	$X = A / B$ A = Mean response time measured by mean response time B = Specified target response time

ID / Name	Description	Measurement function
PTb-3-G, **Mean turnaround time**	What is the mean time taken for completion of a job or asynchronous process?	$X = \sum_{i=1\ to\ n} (B_i - A_i) / n$ A = Time of starting a job i B = Time of completing the job i n = Number of jobs measured
PTb-4-G, **Turnaround time conformance**	How well does the turnaround time meet the specified targets?	$X = A / B$ A = Mean turnaround time measured by Mean turnaround time B = Target turnaround time specified

PTb-5-G, **Throughput conformance**	How well does the throughput meet specified targets?	$X = \left(\sum_{i=1\ to\ n} (B_i / A_i) / n \right) / C$
		A = Number of jobs completed during the observation time.
		B = Observation time period
		C = Target throughput specified
		n = Number of observations

2. **Resource utilization measures.** Resource utilization measures are used to assess the degree to which the amounts and types of resources used by a product or system when performing its functions meet requirement. There are five sub-characteristics of resource utilization measures. The description of each sub-characteristics and measurement function detail in Table 2.4.

Table 2.4 Resource utilization measures

ID / Name	Description	Measurement function
PRu-1-G, **Mean Processor utilization**	How much processor time is used to execute a given set of tasks compared to the l operation time?	$X = \sum_{i=1\ to\ n} (A_i / B_i) / n$ A i = Processor time actually used to execute a given set of tasks

ID / Name	Description	Measurement function
		Bi = Operation time to perform the tasks in observation period i n = Number of observations
PRu-2-G, **Mean memory utilization**	How much of memory is used to execute a given set of tasks compared to the available memory?	$X = \sum_{i=1\ to\ n} (A_i\ /\ B_i)\ /\ n$ A i = Size of memory actually used to perform a given set of tasks Bi = Size of memory available to perform the tasks n = Number of task sets measured
PRu-3-G, **Mean I/O devices utilization**	How much of I/O device busy time is used to perform a given set of tasks compared to the I/O operation time?	$X = \sum_{i=1\ to\ n} (A_i\ /\ B_i)\ /\ n$ A i = Duration of I/O device(s) busy time to perform a given set of tasks Bi = Duration of I/O operations to perform the tasks n = Number of events measured
PRu-4-S, **Storage utilization**	How much of the available secondary storage is used to perform a given set of tasks?	$X = A\ /\ B$ A = Amount of secondary storage actually required to perform a given set of tasks? B = Amount of secondary storage available to the tasks
PRa-5-S, **Bandwidth utilization**	What proportion of the available bandwidth is utilized?	$X = A\ /\ B$ A = Bandwidth of transmission actually measured average over time B = Bandwidth capacity available

3. **Capacity measures.** Capacity measures are used to assess the degree to which the maximum limits of a product or system parameter meet requirements. There are three sub-characteristics of capacity measures. The description of each sub-characteristics and measurement function detail in Table 2.5.

Table 2.5 Capacity measures

ID / Name	Description	Measurement function
PCa-1-G, **Transaction processing capacity conformance**	How many concurrent transactions can be processed at any given time against the specified target?	$X = \left(\sum A_i / B\right) / C$ A_i = Number of active transactions at instant i B = Total operation duration C = Required transaction processing capacity per unit of time specified
PCa-2-G, **User access capacity conformance**	How many users can access the system simultaneously at a certain time against the specified target?	$X = A / B$ A = Number of users simultaneously access the system at a certain time B = Required user access capacity specified
PCa-3-S, **User access increase conformance**	How many users can be added successfully per unit time as compared to the required rate of increase in users?	$X = A / B$ A = Actual number of users successfully added per unit time B = Number of users expected to increase per unit time

In this study, we use definition and terminology based on ISO/IEC 25023 - Measurement of system and software product quality (ISO/IEC, 2011b) and ISO/IEC 25010 - System and software quality models (ISO/IEC, 2011a) as international standard quality model and measurements guideline for performance efficiency.

2.4 Refactoring

In this section, we analyze the concept of refactoring from various perspectives. We focus on definition of refactoring with design pattern. We summarize core concept of refactoring given in (Fowler, Beck, Brant, Opdyke, & Roberts, 1999) (Suryanarayana, Samarthyam, & Sharma, 2014) (Opdyke, 1992) in order to give the fundamentals of refactoring techniques

2.4.1 Core Concepts of Refactoring

The term refactoring was originally introduced by Opdyke in his PhD dissertation to formally explain the behavior-preserving transformation can be made on exiting code (Opdyke, 1992). In software evolution context, refactoring is a reengineering technique or the process of changing a software system that aims at reorganizing a program to improve its quality without changing its external behavior (Society et al., 2014). Martin Felwer defined on his book (Fowler et al., 1999) (code) refactoring as the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal quality structure.

Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. And Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behavior (Fowler et al., 1999). A refactoring aim to improve a certain quality of system while respect others. Refactoring means improving the design of software without altering its noticeable behavior, developer do not add any new requirement features during the process of refactoring, i.e. they do not do any fixes bug of changes anything about software that would be detect by the software user. Instead, only the internal structure of the technology design of the software is changed (Martin Lippert, 2006).

It is a disciplined way provides a technique for cleaning up code in a more efficient and controlled manner that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written. In our current understanding of software development, we believe that we design and then we code. A good design comes first, and the coding comes second. Over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. The code slowly and create maintainable problem and effect the performance efficiency of the system. Refactoring is the opposite of this practice. With refactoring, you can take a bad design, chaos structure, and rework it into well-designed code.

Refactoring is a tool that can or should be used for several purposes. Such as improves the design of exiting software, gain a better understanding of code, helps you find bugs, helps you program faster, make it easier to add new code, make coding less annoying and so on (Fowler et al., 1999) (Kerievsky, 2004). With refactoring, you find the balance of work changes. You find that design, rather than occurring all up front, occurs continuously during development. You learn from building the system how to improve the design. The resulting interaction leads to a program with a design that stays good as development continues.

Mostly we recognize refactoring and classical refactoring technique for low-level code refactoring that focusing on code level transformation in order to reconstruct of anomalies structures. Knowing how to do refactoring does not mean knowing when to do refactoring. Deciding when to start refactoring, and when to stop refactoring is important as knowing how to operate the mechanics of a refactoring. To identify when to apply refactoring, usually use Design Smells (Suryanarayana et al., 2014) and Code Smells, a code smells represent indicators for source code issues such as duplicated code, long method, large class and so on (Fowler et al., 1999) to indicate which part of the system can or should refactor.

2.4.2 Design Smell and Code Smell

Design smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality (Suryanarayana et al., 2014). In other words, a design smell indicates a potential problem in the design structure.

There are various causes of design smells can create bad design and design structure of the system; violation of design principle, inappropriate use of patterns, language limitations, procedural thinking in OO, viscosity, non-adherence to best practices and process (Suryanarayana et al., 2014).

Clearly, smells significantly impact the design quality of a piece of software. It is therefore important to find, analyze, and address these design smells. Performing refactoring is the primary means of repaying process of repaying technical debt. (Ganesh, Sharma, & Suryanarayana, 2013) provide classification of smell that serve design principle. They classified and grouped design smell into four major elements of the object model are abstraction, encapsulation, modularization, and hierarchy. Figure 2.3 illustrates classification scheme and the naming scheme for smells covered.

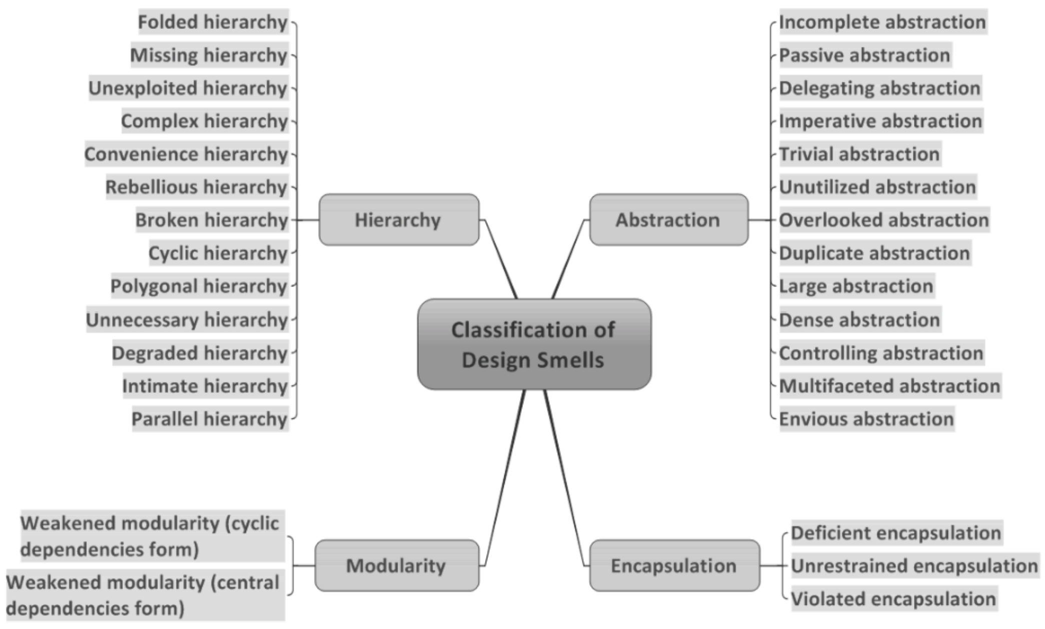


Figure 2.3 Classification of design smells

The most common design problems result from code that are duplicated, unclear and complicated. The coding smells described by Martin Felwer and Kent Beck (Fowler et al., 1999) and Joshua Kerievsky (Kerievsky, 2004) provide a useful way to identify a design problem and find associated refactoring to help fix the problem. They grouped the coding smell into catalog well define, there are various situation that can possible create code smell and they provide catalog of code smell that target problem occurs everywhere, in methods, classes, hierarchies, packages (namespaces, modules), and entire systems. As show in Table 2.6, lists the smells and some refactoring to consider when you want to remove the smells.

Table 2.6 Lists of smells and refactoring associated

Smell	**Refactoring**
Duplicated Code	Form Template Method Introduce Polymorphic Creation with Factory Method Chain Constructors Replace One/Many Distinctions with Composite Extract Composite Unify Interfaces with Adapter Introduce Null Object
Long Method	Compose Method Move Accumulation to Collecting Parameter Replace Conditional Dispatcher with Command Move Accumulation to Visitor Replace Conditional Logic with Strategy
Conditional Complexity	Replace Conditional Logic with Strategy Move Embellishment to Decorator Replace State-Altering Conditionals with State Introduce Null Object
Primitive Obsession	Replace Type Code with Class Replace State-Altering Conditionals with State Replace Conditional Logic with Strategy Replace Implicit Tree with Composite Replace Implicit Language with Interpreter Move Embellishment to Decorator Encapsulate Composite with Builder
Indecent Exposure	Encapsulate Classes with Factory
Solution Sprawl	Move Creation Knowledge to Factory
Alternative Classes with Different Interfaces	Unify Interfaces with Adapter
Lazy Class	Inline Singleton
Large Class	Replace Conditional Dispatcher with Command Replace State-Altering Conditionals with State Replace Implicit Language with Interpreter
Switch Statements	Replace Conditional Dispatcher with Command Move Accumulation to Visitor

Smell	**Refactoring**
Combinatorial Explosion	Replace Implicit Language with Interpreter
Oddball Solution	Unify Interfaces with Adapter

2.4.3 Refactoring Process

Refactoring should be systematically approached in a real-world setting. Normally refactoring is performed in an ad-hoc fashion, resulting in numerous problems. It is, therefore, important to follow a structured approach while refactoring. In this section, we describe a process model called "IMPACT" that provides guidance for systematic refactoring in practice. IMPACT is comprised of the following four fundamental steps that are executed in order These steps are described in the following sub-sections and illustrated in Figure 2.4.

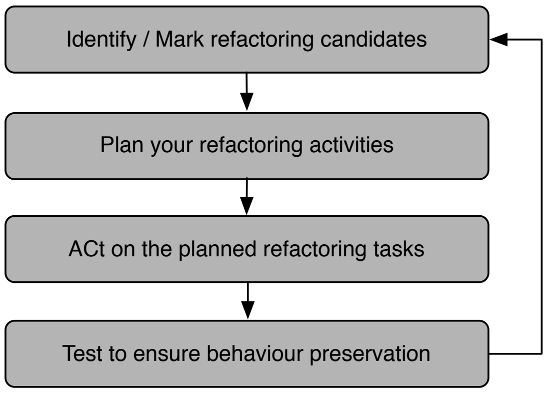

Identify / Mark refactoring candidates
Plan your refactoring activities
ACt on the planned refactoring tasks
Test to ensure behaviour preservation

Figure 2.4 IMPACT refactoring process model.

1. **Identify / Mark refactoring candidates**. The first step in the refactoring process is to analyze the code base and identify refactoring candidates.
2. **Plan your refactoring activities.** analyze their impact, prioritize them, and prepare a plan to address them. Based on the prioritized list of identified smells and their refactoring, an execution plan for the refactoring can be appropriately formulated
3. **Act on the planned refactoring tasks.** Team members can take up planned refactoring tasks and execute them by carrying out the refactoring in the code.

4. **Test to ensure behavior preservation.** Refactoring activity should be followed by automated regression tests to ensure that the behavior post-refactoring is unchanged.

In this thesis, we use refactoring definition and terminology from (Fowler et al., 1999). To be systematic way in refactoring, we do follow IMPACT model in implementation.

2.5 Academic Information System

Academic Information System is an information system with business process for education propose. It consists of various processes and functions handle the education and high education requirement in systematic way.

The Sistem Informasi Akademik (SIA) is an Academic Information System project that have been creating and maintaining by faculty of Informatics Engineering, Institut Teknologi Sepuluh Nopember. It is design based on Module type architecture with support high cohesion and loosely-coupled. It created based on current Java Enterprise Edition (JEE) technology with Model-View-Controller (MVC) architecture, using Spring MVC and Hibernate ORM framework as helper libraries. SIA use Eclipse Virgo and OSGi Framework. The backend web server that use to run SIA is Apache Tomcat. SIA is use PostgreSQL as main database.

Currently SIA consist of six modules, they are Framework Module (Yuhana, Akbar, Agung, & Wijaya, 2016), Domain Module (Yuhana, Akbar, & Nurwantoro, 2015), Pembelajaran Module (Yuhana, SUMINTO, & Anggraini, 2015), Kurikulam Module (Rochimah, Akbar, & AVEROUSI, 2015), Ekivelensi Module (Yuhana, Anggraini, & Alfirdaus, 2015), and Penilain Module (Rochimah, Anggraini, & RAHMAN, 2015). The Framework module is the main based module. The domain module as interoperable of other modules connecting to the Database system. and the rest of modules are responsible for academic service functions. Each module has been built in separately with different objective of requirements. To make all modules collaborate and integrate, (Yuhana et al., 2016) refactored SIA apply HMVC architecture pattern and utilized modularity principle to build integration space together and connected with Domain Module to exchange message, share libraries and functions. The packet diagram is shown overall structure of SIA as illustrated in Figure 2.4.

In this thesis, we use Sistem Informasi Akademik (SIA) as our case study to conduct experiment of our approach. We select to focus on the critical hot spot problem module through analyzing phase.

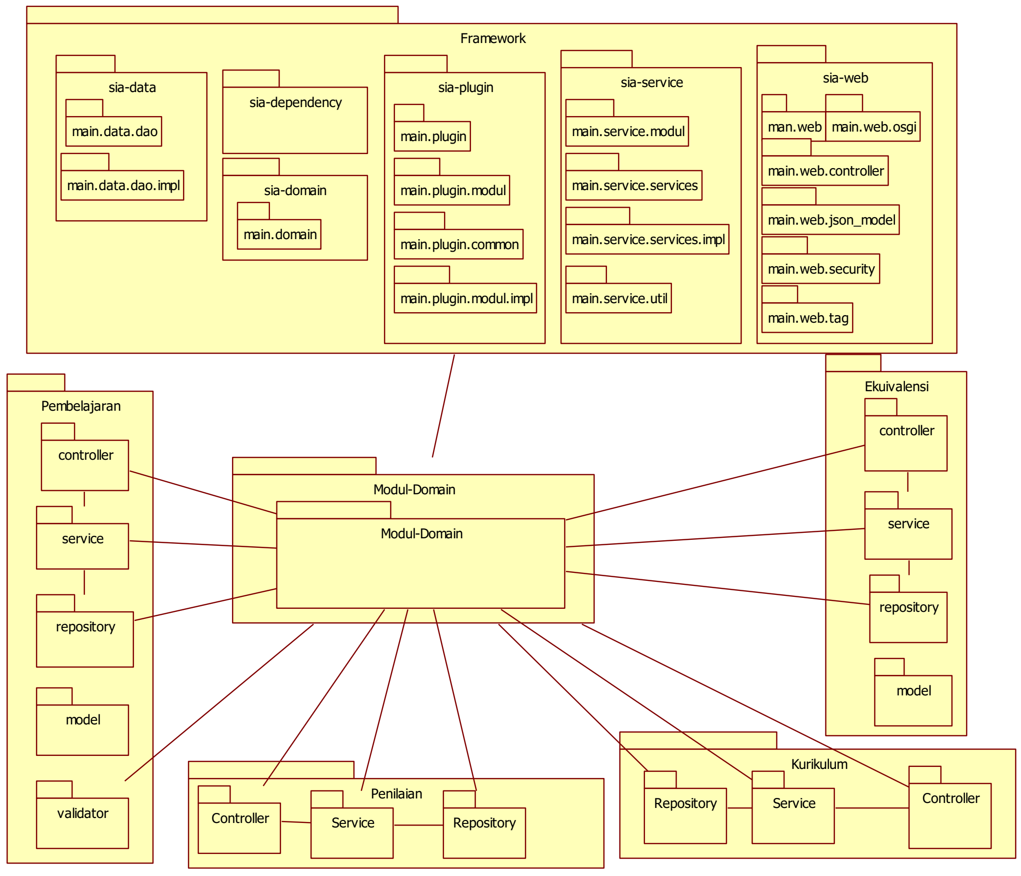


Figure 2.5 Packet Diagram of Sistem Informasi Akademic (SIA)

(This page intentionally left blank)

CHAPTER 3
RESEARCH METHODOLOGY

3.1 Research Methodology

Our research methodology has five phases; problem analysis, solution design, solution implementation, solution validation, and report results (see Figure 3.1).

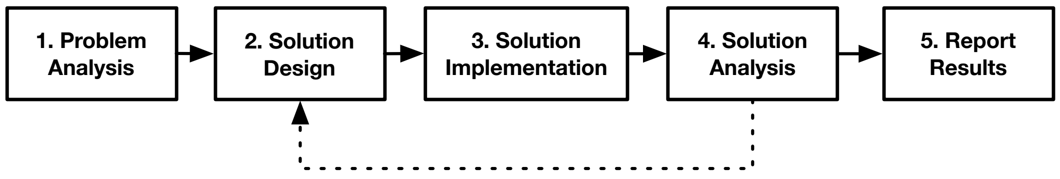


Figure 3.1 Research Methodology

3.2 Problem Analysis

In the first phase, we solve knowledge problem. We want to understand what the current performance efficiency measurement are, what the current refactoring approaches with design pattern principles are, and what the deficiencies of the current approaches are. For these purposes, we analyze the literature about refactoring, design patterns, especially "Gang of Four" design patterns, Object-Oriented approach, and Software Quality from different research areas to discover possible problems in current refactoring approach through design pattern concept in software development design phase. Especially in performance efficiency quality attribute and measurement.

We analysis the case study to see feasibility of improvement in software design for targeting measure performance efficiency. As SIA have been created by group of students of Informatics Engineering Department of Institut Teknologi Sepuluh Nopember (ITS). The application has been considered on function requirements. Developer used tools support in developing the system. The tools were given a rapid development and well-suited in continue development. The code structure that have been generated and coded in Java and OO approach. Developers used OO approach in developing the system by force of Java development environment and framework used but they were lack of realize OO design and design patterns principle. Without considered OO design principle and best practice of existing design patterns, the system did not appropriate maximum apply OO technology and best exiting solution in designing and developing, especially in low level design and coding. The result of this unaware, the system structure and code detail become ambiguous for continue development.

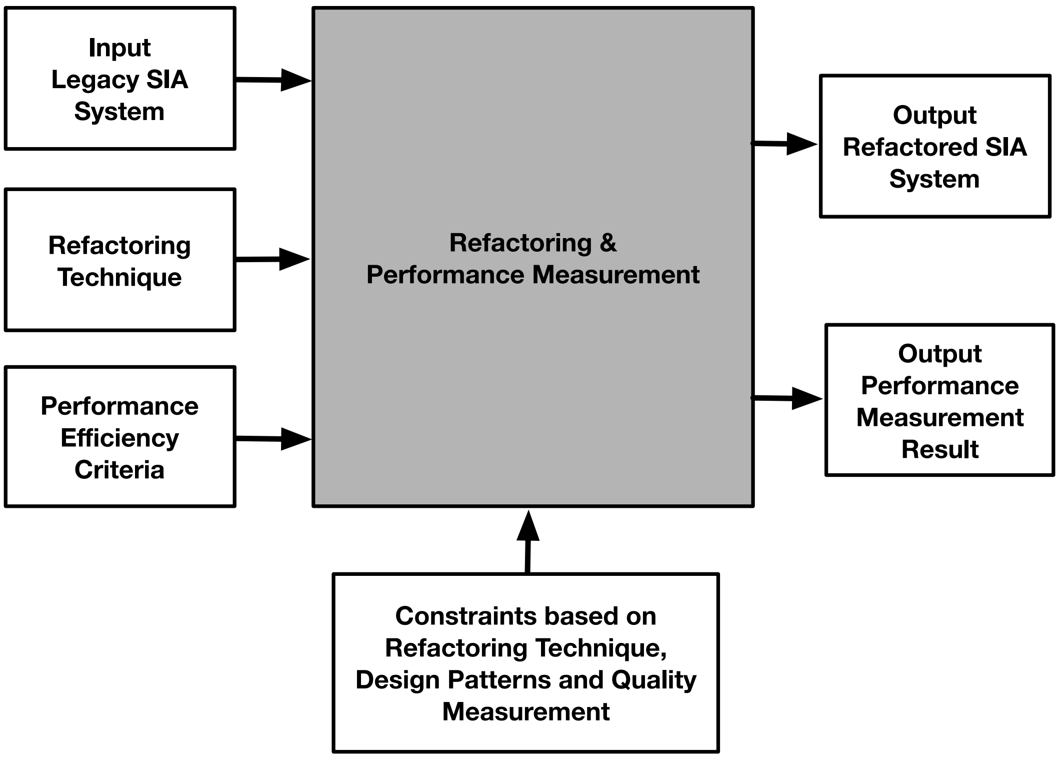
Application developer should consider the impact of applied these kinds of principles and tools as in performance efficiency quality attribute, a factor that always affect continue development. Performance of an application have to or must measure, especially when applied refactoring process and design patterns in order to achieve better result on performance quality criteria such respond time, throughput and resource use by new refactored system. It should consider the impact of applied these kinds of principles and tools for better software design.

3.3 Solution Design

In the second phase, the results of the first phase are used to design a new solution. We solve a particular design problem by provide an approach with case study implementation, we refactor the system using design patterns with respect to performance efficiency software quality attributes. Our goal is to measure the impact of design patterns through refactoring process of the system with rely on quality attribute of software.

3.3.1 Overview of the Approach

Our approach given idea of the how refactoring and performance efficiency measurement can be performed through several constraints. Figure 3.2 gives the overview of the approach.



Input Legacy SIA System
Refactoring Technique
Performance Efficiency Criteria
Refactoring & Performance Measurement
Output Refactored SIA System
Output Performance Measurement Result
Constraints based on Refactoring Technique, Design Patterns and Quality Measurement

Figure 3.2 Overview of the Approach

1. **Input Legacy SIA.** We use Academic Information System or (Sistem Informasi Akademik - SIA, in Indonesia language) as our case study. We select a module from all available to study and do implementation.
2. **Refactoring Technique.** Refactoring process, method is applied (Section 3.3.4).
3. **Performance Efficiency Criteria.** We use performance efficiency international standard criteria, time behavior, resource utilization and performance efficiency compliance to identify and verify the performance of proposed approach (Section 3.3.5).
4. **Refactoring and Performance Measurement.** We use refactoring technique through the selected of design patterns in refactor and modify internal structure of the system, and performance measurement is used to assess the performance relative to amount of resources used under stated conditions of the target system.

5. **Output Refactored SIA.** The result of the processes will give a candidate refactored SIA with improve the internal structure and result better in performance validation. From this output, we compare the legacy system with candidate refactored system to investigate the impact of the refactoring process using design pattern in performance efficiency attribute.
6. **Output Performance Measurement Result.** The result of performance efficiency produced in this step. The performance factors are measured and produced the output to use in analysis. This step give an evidence result of the design pattern and refactoring process affect application performance. The affect may positive or negative improve of the application performance.
7. **Constraints.** Several constraints contain in the processes of the approach, such as refactoring technique, design pattern technique, legacy system and performance quality measurement, tools support.

We use tool in supporting and illustrating the feasibility of our approach in an example.

1. **Tool support.** We describe the use of tool support. Several tools use to support the approach. We use Eclipse as our main Integration Development Environment (IDE). Visual Paradigm version 13.2 and ObjectAid UML Explorer version 1.1.4 for tools in reverse engineering from source code to diagram and from diagram to code, Eclipse Metrics plugin version 1.3.6 (Sauer, n.d.) as specific detail complexity measurement of the system. Apache JMeter version 3.1 (Foundation, n.d.) is load test functional behavior and measure performance application tool and Java Mission Control 5.5 (Corporation, n.d.) for Java Profiling tool, these tools are giving a chance in performing performance efficiency measure according to our quality requirement measurement. And we follow the International Standard, ISO/IEC 25023 – Measurement of system and software product quality, in measuring specific focus on performance efficiency factor.
2. **Running Example.** We illustrate the approach with case study. Our selected case study is a SIA module.

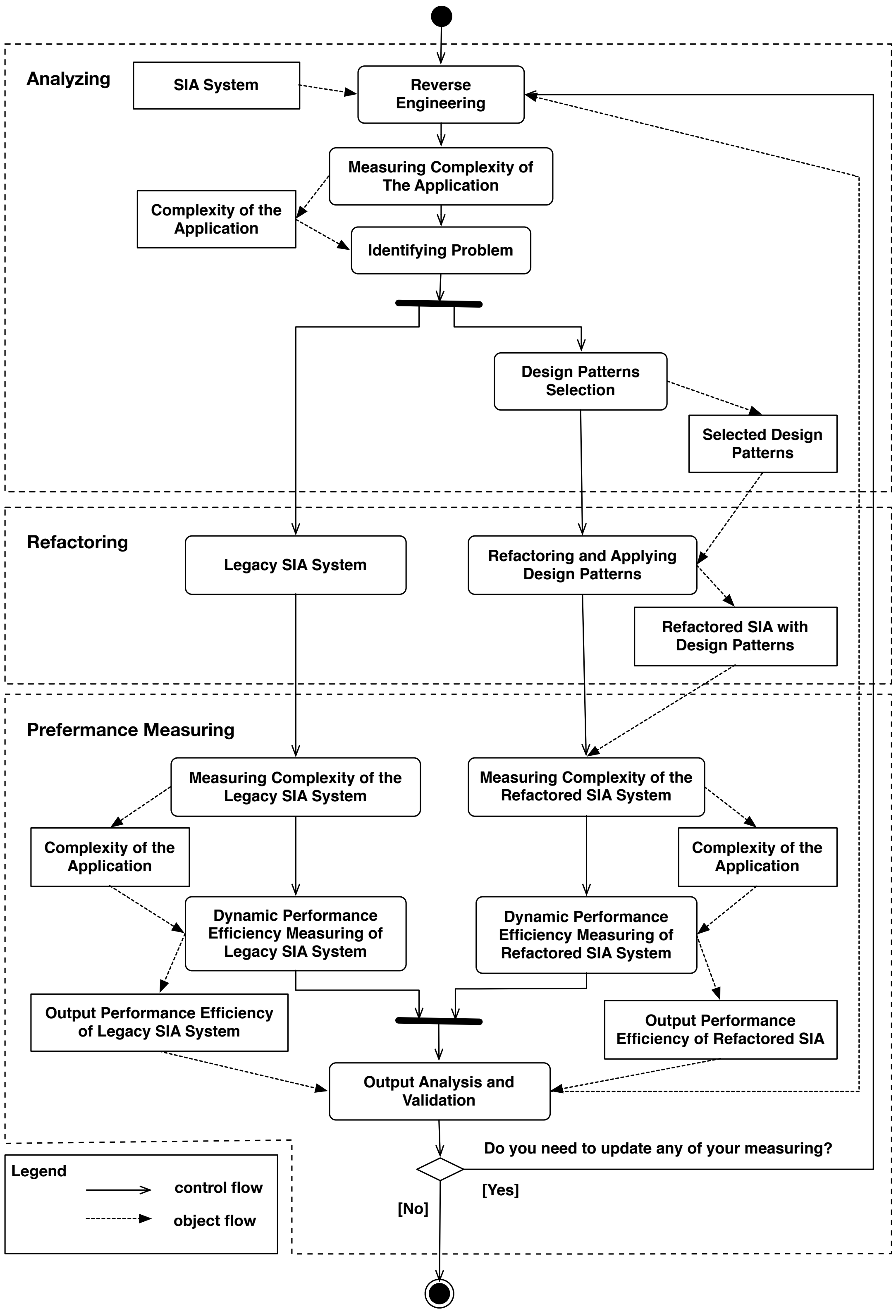

Analyzing
SIA System
Reverse Engineering
Measuring Complexity of The Application
Complexity of the Application
Identifying Problem
Design Patterns Selection
Selected Design Patterns
Refactoring
Legacy SIA System
Refactoring and Applying Design Patterns
Refactored SIA with Design Patterns
Prefermance Measuring
Measuring Complexity of the Legacy SIA System
Measuring Complexity of the Refactored SIA System
Complexity of the Application
Complexity of the Application
Dynamic Performance Efficiency Measuring of Legacy SIA System
Dynamic Performance Efficiency Measuring of Refactored SIA System
Output Performance Efficiency of Legacy SIA System
Output Performance Efficiency of Refactored SIA
Output Analysis and Validation
Do you need to update any of your measuring?
[Yes]
[No]
Legend
control flow
object flow

Figure 3.3 Phases of the proposed approach

3.3.2 The Process of Proposed Approach

The proposed approach is structured in three fundamental phases. Figure 3.3 gives a UML activity diagram of the process. The process in Figure 3.3 consists of the following activities:

3.3.3 Analyzing

Analyzing the system to gain an idea about the complexity of the study application. The analysis step consists as below:

1. **Reverse Engineering.** We use object-oriented reverse engineering technique (Demeyer et al., 2002) focusing on as the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. We use several sources of information while reverse engineering, such as read the existing documentation, read the sources code, run the software, use tools to generate high-level view of the sources code. These sources of information help a lot in analyzing, re-documenting and identifying potential problems of the software application. The result of this activity details of the system such as Architecture View and Class Diagram.
2. **Measuring Complexity of the Application.** This activity takes the SIA and the Object-Oriented metric (Henderson-Sellers, 1996; Sauer, n.d.). This activity does measuring complexity of the application. This activity produces summarize feasibility and detail of the application with Total Lines of Code (TLOC), Number of Classes (NOC), Number of Methods (NOM), Number of Packages (NOP) and calculate the McCabe Cyclomatic Complexity (MCC), as well-known complexity metric measure for the complexity of the application.
3. **Identifying Problem.** The result from steps above, Reverse Engineering and Measuring Complexity of the Application use to determine and identify feasible problems occur in the application.
4. **Design Patterns Selection.** The analyzing result from previous activities given signs of Code Smell and Design Smell (Fowler et al., 1999; Suryanarayana et al., 2014) issue related to the legacy software application. The design patterns is selected based on "Gang of Four" design patterns categories. The "Gang of Four" classified design patterns purpose into three categories: creational,

structural and behavioral. The choice of design patterns is not arbitrary; an architect of a software system can face this kind of choice during the design process. The selection of design pattern is selected based on the problem facing in specific context and judge by group of expertise and researchers. The result of this activity is the suitable selected design pattern that going to adapt and implement into the SIA in refactoring process.

3.3.4 Refactoring

The process of refactoring is an activity change made to 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 legacy code. When we refactor, we relentlessly restructure and modify the code to improve its design. Such improvements involve with apply suitable selected design patterns aims at changing as small part of code design structure or as large as unifying two hierarchies. These activities consist of following:

1. **Refactoring and applying design patterns.** This activity to apply design pattern to legacy system through refactoring technique. The selected design pattern chooses to study and refactor to the system. In this activity, we follow the IMPACT refactoring process model. There are four fundamental steps: Identify and Mark refactoring candidates, Plan your refactoring activities, Act on the planned refactoring tasks, and Test to ensure behavior preservation. Figure 3.4 gives a UML activity diagram of the refactoring process. The detail description of each step as follow:

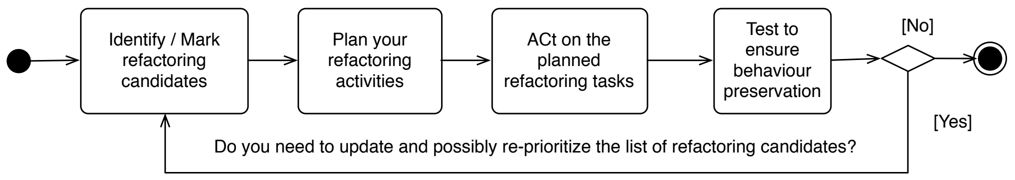

Identify / Mark refactoring candidates
Plan your refactoring activities
ACt on the planned refactoring tasks
Test to ensure behaviour preservation
[No]
[Yes]
Do you need to update and possibly re-prioritize the list of refactoring candidates?

Figure 3.4 Activities of IMPACT refactoring process model

a. **Identify and Mark refactoring candidates**. This activity of the refactoring process is to analyze the code base and identify refactoring candidates. The case study can carry out manual code and design review to find smells and determine candidate for refactoring. Manual reviews are more effective and less error-prone since they can consider and exploit domain knowledge, the context of the design, and design expertise more effectively.
b. **Plan your refactoring activities.** In this activity, once we identify smells using complexity measurement and manual code and design review (Section 3.3.3, Reverse Engineering and Measuring Complexity of the Application), it is important to analyze their impact, prioritize them, and prepare a plan to address them. To analyze the impact of a smell, consider factors such as severity, scope and interdependence. After analyzing the impact of the identified smells, prioritize them based on the following factors; the potential gain after removing the smell, available time, availability of tests for the target modules. Based on the prioritized list of identified smells and their refactoring, the design pattern selection process starts to select suitable design based on identified smell. And an execution plan for the refactoring can be appropriately formulated
c. **Act on the planned refactoring tasks.** This activity can take up planned refactoring tasks and execute them by carrying out the refactoring in the code. In this process, we also use automated refactoring support provided by IDEs to carry out the refactoring tasks.
d. **Test to ensure behavior preservation.** This activity is very important step in refactoring process. We refactoring activity should be followed by automated regression tests to ensure that the behavior post-refactoring is unchanged. We first tests for the entity that needs to be refactored, then refactor the entity, and finally test it to verify the behavior.

The result of refactoring process gives a refactored SIA with applied design pattern properly. On the opposite side of refactoring and applying design patterns in refactoring process phase, legacy SIA, we do nothing change on the system. The legacy system remains everything same for the purpose of comparison. The results of this process are refactored and legacy SIA. These outputs will use in the next step of the approach.

3.3.5 Performance Measuring

Evaluation is the analytical step of the process. In this step, the current measurement is compared against previous measurements or against expected values. The individual statistics of the two measurements are carefully compared and the differences analyzed. If it determines that its purpose goals have been met, then the goals is completed. If not, it may continue on to modification, the next step of the process. The measurement here adheres to dynamic measure. It means behavior and test environments of the system is variable and flexible.

1. **Measuring Complexity of the Application.** This activity, once again complexity measurement take control. This activity produces summarize and detail of the application of both refactored and legacy SIA with Number of Packages (NOP), Number of Classes (NOC), Number of Interfaces (NOI), Number of Attributes (NOF), Number of Methods (NOM), Total Lines of Code (TLOC). And calculate the McCabe Cyclomatic Complexity (MCC), Weighted Methods per Class (WMC), Lack of Cohesion of Methods (LCOM), Efferent Coupling (CE), and Afferent Coupling (CA). We also do modification comparison between refactored and legacy SIA to determine the changed point and internal structure that have been refactored.
2. **Dynamic Performance Efficiency Measuring.** This activity uses to measure both refactored and legacy SIA. We utilize core performance testing activities defined in (Meier, Farre, Bansode, Barber, & Rea, 2007). There are seven core activities, Identify Test Environment, Identify Performance Acceptance Criteria, Plan and Design Tests, Configure Test Environment, Implement Test Design, Execute Tests, and Analyze, Report, and Retest. This measurement measure three type of performance efficiency factors: time behavior, resource utilization and performance efficiency compliance. To achieve this target, we use performance measurement and profiling tools support. We use Apache JMeter for measuring time behavior and performance efficiency compliance in load test functional behavior and measure performance application. JMeter tool can send a number of requests that simulate an activities user of the application. We use Java Mission Control (JMC), Java Profiling tools for measuring

resource that have been used in specific given time. By collecting information on the response time and content, and resource use, it is possible to calculate all defined parameters measure. The test data is generated in CSV and XML file format. These tools are giving a chance in measure performance efficiency according to criteria quality requirement measurement specified. Figure 3.5 gives a High-Level sequence diagrams of the processes of JMeter activities and Java Mission Control in capture resource use.

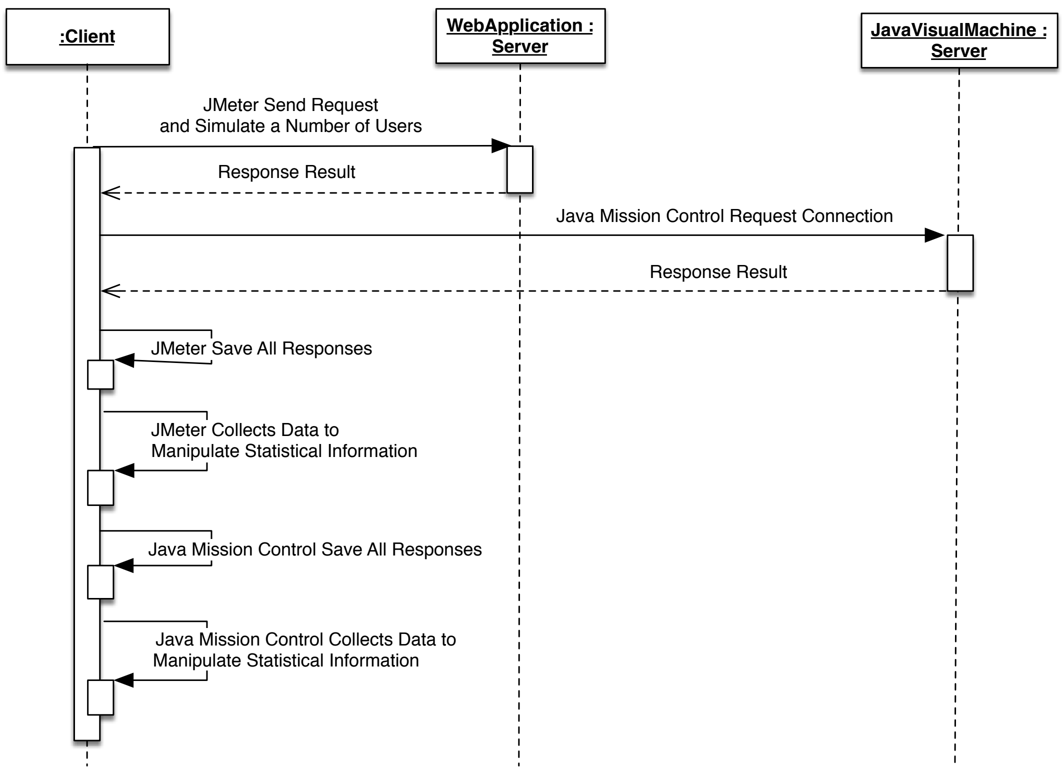

:Client
WebApplication : Server
JavaVisualMachine : Server
JMeter Send Request and Simulate a Number of Users
Response Result
Java Mission Control Request Connection
Response Result
JMeter Save All Responses
JMeter Collects Data to Manipulate Statistical Information
Java Mission Control Save All Responses
Java Mission Control Collects Data to Manipulate Statistical Information

Figure 3.5 Sequence diagram of JMeter and Java Mission Control interaction

The test scenario uses for performance efficiency measurement need to reflect the main functionality user activity of the case study system. In principle, the test activities represent the following data operation: querying, creation, removal, and update, with data querying being the most prevalent activity. The test scenario includes activities typical for this kind of application:

1. listing student details,
2. generating and viewing report,
3. adding users, and

4. removing users from the application.

There are a number of performance efficiency parameters measure during each test. Primarily, we are gathered parameters that directly related to application performance and support by available tools. The parameters measure includes:

1. Time Behavior Measures

Mean response time, this measurement function use to measure mean time taken by the system to respond to a user action, where A_i is time between a user's request and a system's response, n is total number of response events measured. The equation is given in Equation (3.1). *Response time conformance*, this measurement function use to measure how well does the system response time meet the specified target, where A is mean response time result of Equation (3.1), and B is specified target response time. The equation use to measure is given in Equation (3.2).

$$MeanResponseTime = \sum_{i=1\ to\ n} (A_i) / n \tag{3.1}$$

$$ReponseTimeConformance = A / B \tag{3.2}$$

Throughput conformance, this measurement function use to measure how well does the throughput meet specified targets, where A is number of tasks completed during the observation time, B is observation time period, C is target throughput specified, and n is number of observations. The equation use to measure is given in Equation (3.3)

$$ThroughputConformance = \left(\sum_{i=1\ to\ n} (B_i / A_i) / n \right) / C \tag{3.3}$$

The data source use for calculate time behavior characteristic able to obtain from execute the web performance test through client-server message request and response using HTTP Request and HTTP Response protocol.

2. Resource Utilization Measures

Mean Processor utilization, this measurement function use to measure how much processor time is used to execute a given set of tasks compared to the operation time, where A_i is processor time actually used, B_i is operation time to perform the tasks in observation period i, n is the number of observations. The equation use to measure is given in Equation (3.4). *Mean memory utilization,* this measurement function use to measure how much of memory is used to execute a given set of tasks compared to the available memory, A_i size of memory actually used to perform a given set of tasks, B_i is size of memory available to perform the tasks, n is number of task sets measured. The equation use to measure is given in Equation (3.5)

$$MeanProcessorUtilization = \sum_{i=1\ to\ n} (A_i \ / \ B_i) \ / \ n \tag{3.4}$$

$$MeanMemoryUtilization = \sum_{i=1\ to\ n} (A_i \ / \ B_i) \ / \ n \tag{3.5}$$

The data source use for calculate resource utilization is take from capturing resource use during perform web performance testing. Java Mission Control allow us to detailed low-level runtime information of processor and memory use of Java Virtual Machine (JVM) and the Java application. The resource utilization enables us to collect and analyze data from Java applications running locally and remotely.

3. Capacity Measures

Transaction processing capacity conformance, this measurement function use to measure how many concurrent transactions can be processed at any given time against the specified target, where A_i is number of active transactions at instant given i, B is total operation duration, and C is required transaction processing capacity per unit of time specified. The equation use to measure is given in Equation (3.6).

$$TransactionCapacity = \left(\sum A_i \ / \ B\right) \ / \ C \tag{3.6}$$

The data source that used to calculate capacity conformance is getter from execute the web performance test through client-server message request and response using HTTP Request and HTTP Response protocol. We simulate transaction process and user access and continues increase the number of transaction and process to investigate how far the JVM and Java application can handle the specified load test.

3. **Output Analysis and Validation.** This activity analyzes and validate the measurement result for performance efficiency of refactored and legacy SIA system (Section 3.5). The result of this step provides the source of data to analyze and investigate the impact of design pattern through refactoring process. The process will show the impact of design pattern either positive of negative in performance measurement criteria.

3.4 Solution Implementation

In this phase, the solution design is implement in this step. The implementation follows the step processes that given in solution design start to initializing and experimental. The process including of analyzing, refactoring and evaluation measurement.

In order to simulate the activities of a number of web application users, a load generator tool is used. The tool chosen is Apache's JMeter. JMeter is able to send HTTP requests to an application according to a predefined scenario. The tool supports the simulation of multiple concurrent users, as well as the ability to specify intensity of the test. The tests are conducted in series, in one series included several simulations. In the first test stage, 5 concurrent users are simulated. In the second test stage, 10 users are simulated, and in subsequent test stages, the number of simulated users are always increased by 10 users. Finally, the test simulated reach maximum concurrent users that application and server can handle to accessing the test application. Each round is repeated 3 times to ensure that the results are meaningful and reliable. We use Java Mission Control tool for capture resource data use in a particular given time in order to collect and detail information about Java Application that use CPU and Memory during execute time running on the Java Virtual Machine (JVM).

The performance efficiency parameters are giving in (Section 3.3.5). These set of parameters allow us to thoroughly observe how the application's behavior change depending on the number of users for each of the investigate design variants used for each iteration. The approach implementation conducts in a controlled environment. The controlled environment gives accurate result and minimize noise during run the test. The test environment consists of two machines located in a local network, the server and the client side machine. The test environment as show in Table 3.1.

Table 3.1 Test Environment Specification

Environment	**Specification**	**Server**	**Client**
Hardware	Processor	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz 64bits	Intel Core i7 2GHz
	RAM	7855MB	16GB
	HDD	500GB	500GB
	Network Card	1Gbit/s	1Gbit/s
Software	Operating System	Ubuntu 14.04.3 LTS (GNU/Linux 3.19.0-42-generic x86_64)	Macintosh; Intel Mac OS X 10.11
	Development Tools	(Java HotSpot(TM) 64-Bit Server VM (build 24.80-b11, mixed mode)) with Java(TM) SE Runtime Environment (build 1.7.0_80-b15), Virgo Server 3.6.4 and PostgreSQL 9.4	Eclipse 4.4 with Spring Tool Suite (STS) 3.7.3 installed, Visual Paradigm 13.2, ObjectAid UML Explorer 1.1.4, Hibernate Framework 4.2
	Performance Testing Tools		JMeter 3.1 and Java Mission Control 5.5

3.5 Solution Analysis

In this phase, we analyze our solution by investigating its availability for the problems discovered in the problem analysis phase. This is a knowledge problem since we want to gain knowledge about the properties of our solution, and the relation between the solution and the problems. The outcome of the solution analysis phase is fed back to the solution design phase in order to improve the solution. In this phase analysis process of the hypothesis will analyze the solution design based on given criteria. The analysis result will achieve when hypothesis have been answered. There are two main criteria of this thesis, either positive or negative impact of design patterns in refactoring process in improvement of design structure of code. To achieve this goal, we use particular performance efficiency measures (Section 3.3.5) to measure both systems, legacy and refactored systems. These measurement result provides the mean and median values of the measured timings across all runs instead of reporting only best or worst runs.

We do validation our experiment result by comparing gained result detail between legacy and refactored system to see the differentiate of the systems. We present our results in statistical and graphical presentation based upon the raw experimental data.

3.6 Report Results

In this phase, we do documentation report of obtain result from each step from literature and theory, problem analysis, solution design, solution implementation and solution validation result. The report will be follow the standard guide provided by institute. The report will be useful and future worth for reader who are interesting in the topic focusing area of the thesis cover.

3.7 Preliminary Experiment

In this phase, we do some part of the proposed approach preliminary experiment to ensure in general for propose approach is realize and possible in implementation. Our select case study is *Adapter* patterns. In Figure 3.5 is the legacy code of the application. And Figure 3.6 illustrated the application when refactored using *Adapter* design patterns.

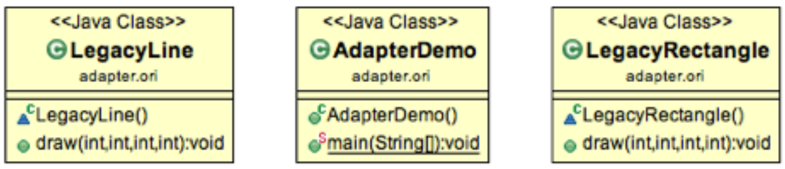


Figure 3.6 Legacy of Line and Rectangle objects

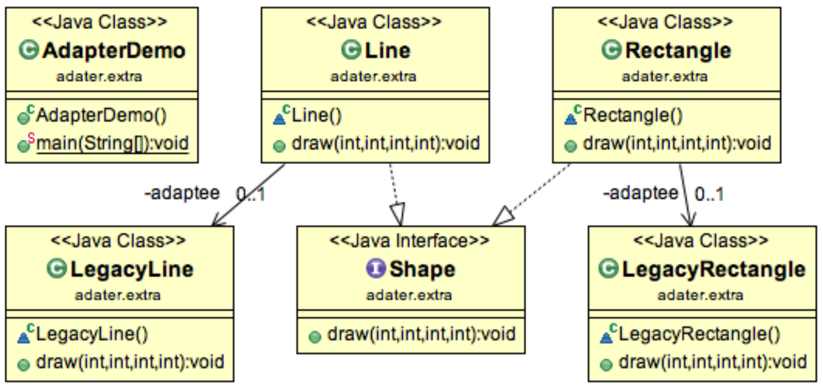


Figure 3.7 Adapter with extra level of indirection

In Table 3.2, the complexity measurement of Total Line of Code is 35 and 52 LOC for legacy and refactored system. The LOC of refactored is increased because of extend to create another extra classes and interfaces. The Number of Classes (NOC) are 3 and 5 classes for legacy and refactored system. The Number of Methods (NOM) are 2 and 4 methods for legacy and refactored. The Number of Interfaces (NOI) is added 1 interface for refactored. The Number of Attributes (NOF) is added 2 attributes for refactored system. The metric of Cyclomatic Complexity measure gains 2 and 1.2 for Mean on both side.

Table 3.2 Complexity of the Application

Complexity of the applications	**Legacy**		**Refactored**	
	Total	**Mean**	**Total**	**Mean**
Lines of Code (LOC)	35		52	
Number of Classes (NOC)	3		5	
Number of Methods (NOM)	2	0.667	4	0.8
McCabe Cyclomatic Complexity (VG)		2		1.2
Number of Interfaces (NOI)			1	
Number of Attributes (NOF)			2	0.4

The complexity measurement has shown that the compared result between refactored application is less complex than original legacy application as present in graph depicted in Figure 3.8 and Figure 3.9.

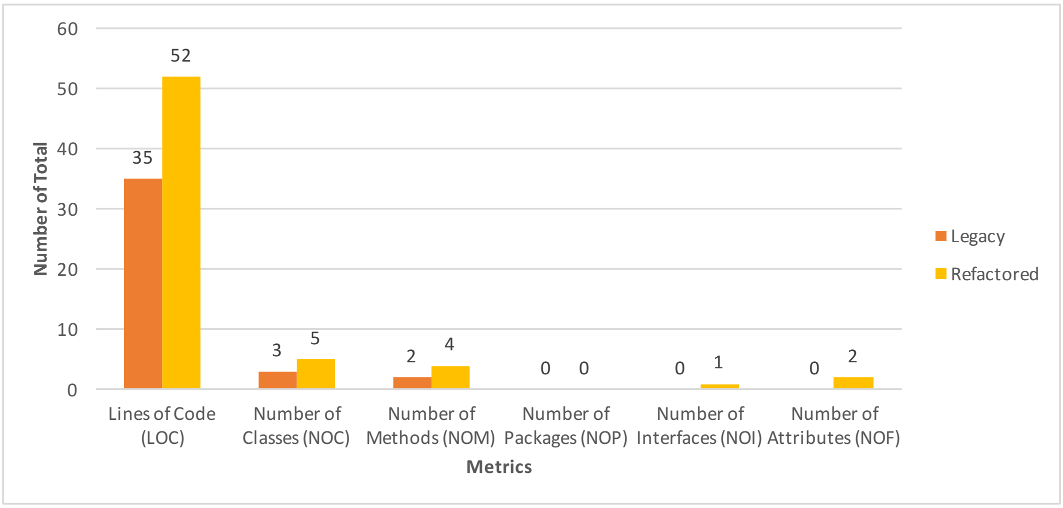


Figure 3.8 Graph representation of number of total

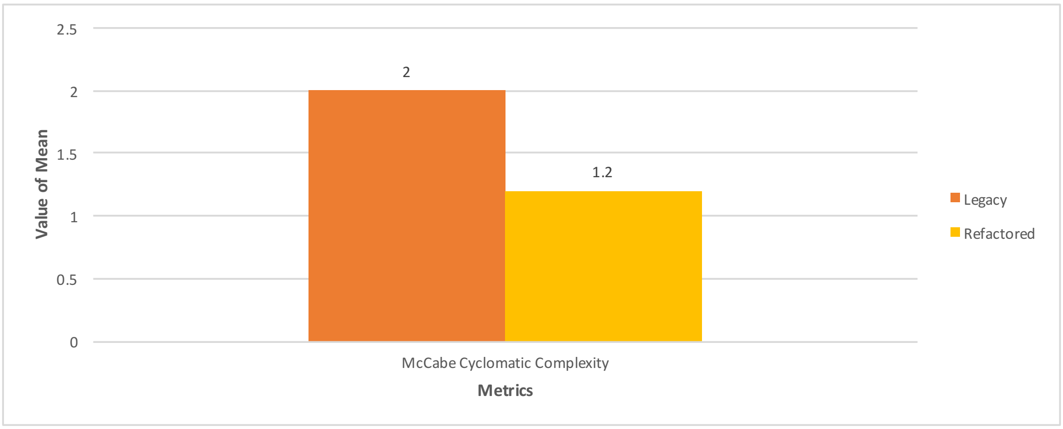


Figure 3.9 Graph representation of complexity of the application

(This page intentionally left blank)

CHAPTER 4
IMPLEMENTATION RESULT

In this chapter, the solution design is implement in this step. The implementation follows the step processes given in solution design start to initializing and experimental. The process including of analyzing, refactoring and performance measuring. The implementation and discussion of the obtain result given in this chapter.

4.1 Analyzing

4.1.1 Reverse Engineering

Academic Information System or (Sistem Informasi Akademik) or SIA for short, is created based on Modularity architecture and implemented Hierarchical Model-View-Controller (HMVC), the pattern decomposes the client tier into a hierarchy of parent-child MVC layers. The repetitive application of this pattern allows for a structured client-tier architecture for reduces dependencies and increases extensibility.

This version of Academic Information System is constructed by team of students and researchers at Department of Informatic Engineering, Institut Teknologi Sepuluh Nopember, Surabaya. The first snapshot version of SIA system consists of four modules, there are Kurikulam Module, Pembelajaran Module, Penilain Module and Ekivelensi Module as illustrated in Figure 4.1

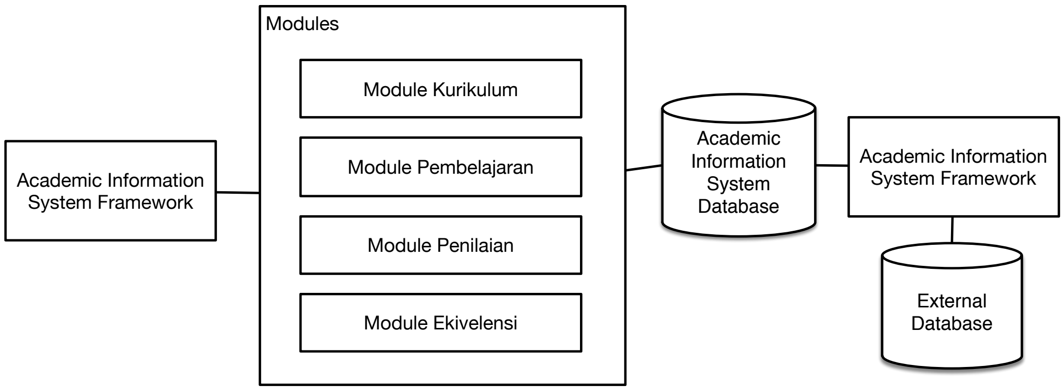

Modules
Module Kurikulum
Module Pembelajaran
Module Penilaian
Module Ekivelensi
Academic Information System Framework
Academic Information System Database
Academic Information System Framework
External Database

Figure 4.1 Academic Information System architecture design

Then, the SIA system was refactoring to advance modularity architecture. The SIA system framework created based on five layers' modularity architecture serving Separation of Concerns principle, there are *Web Layer, Service Layer, Data Layer, Plugin Layer,* and *Domain Layer.* All these layer is built and deploy independently on OSGi framework. Figure 4.2 illustrated UML component diagram architecture of SIA system.

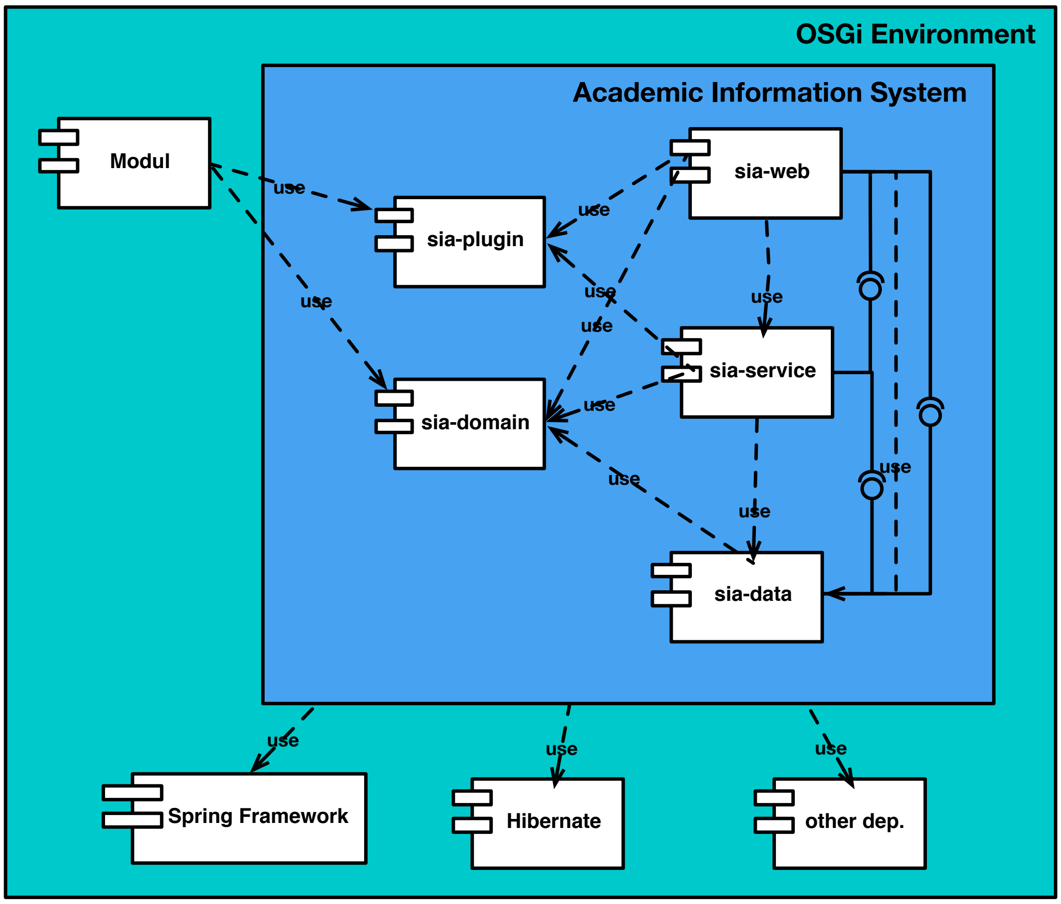


Figure 4.2 Component diagram of Academic Information System Architecture Design

Currently, SIA system that was refactored consist of six modules, there are Framework Module, Domain Module, Pembelajaran Module, Kurikulam Module, Ekivelensi Module, and Penilain Module. The architecture model component designs each module of SIA illustrated in Figure 4.3.

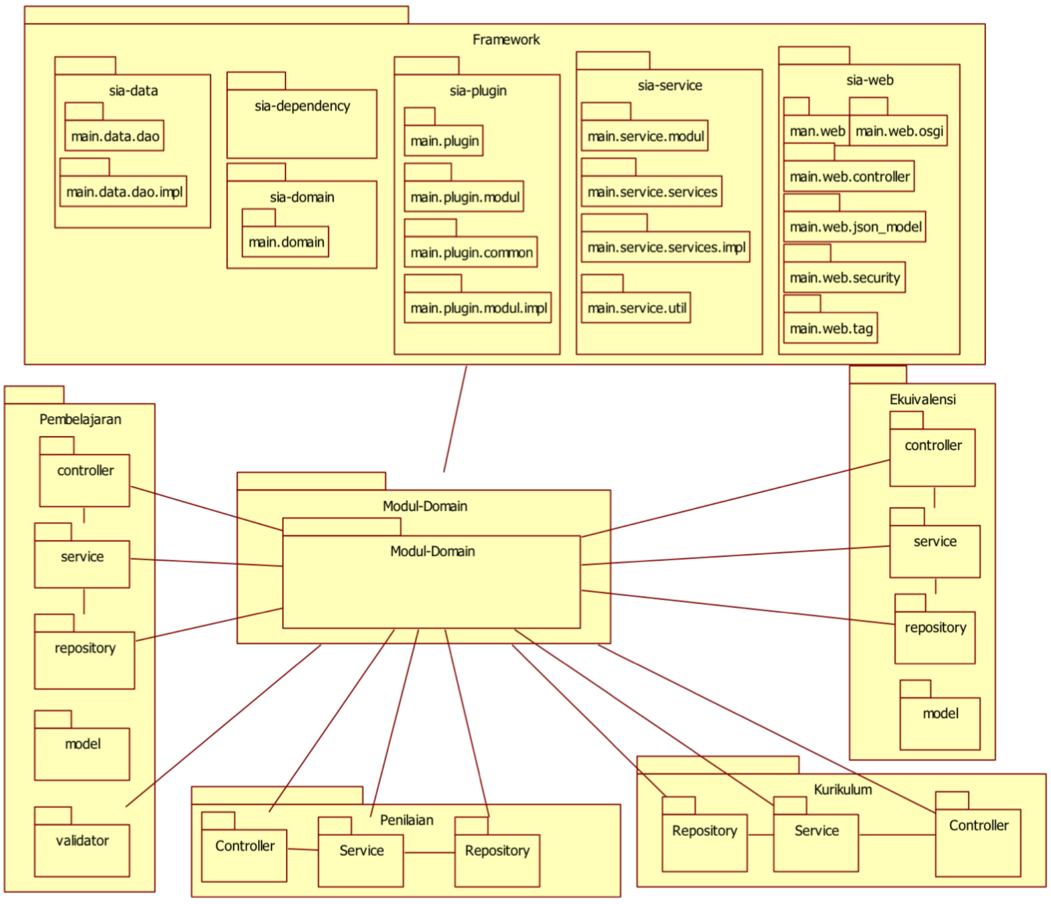


Figure 4.3 Packet Diagram of Sistem Informasi Akademik (SIA)

In our case study research implementation, we utilize experiment the refactoring process on Module Penilaian (Grading Module). The design of Grading Module is illustrated in Figure 4.4. The module architecture is to split the project into several logical layers.

1. Client side (what users see in browsers): UI layer, in our case study system use HTML/JSP page with JSTL and Spring forms
2. Server side: Controller layer (Spring MVC), Service layer (Spring), Repositories (Spring and Hibernate)
3. Data layer: PostgreSQL
4. Model – Java bean classes, which represent application data objects.

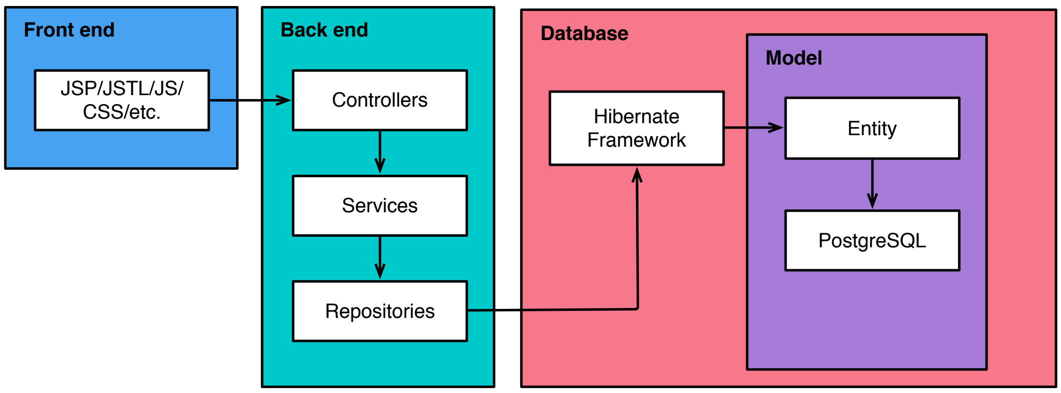


Figure 4.4 Design of Module Penilaian Architecture

Grading Module allows administrator and instructors to submit or change assignment and examination mark, final grades, generate the report, generate student transcript, produce IPS and IPK scores, managing questionnaire for the students in their courses. The SIA grading module present 14 features as list in Table 4.1.

Table 4.1 Lists of Grading module features

No.	Features
1	Managing student learning outcome results
2	Managing the assessment component
3	Viewing teacher questionnaires performance report
4	Viewing the questionnaire report per class
5	Viewing the questionnaire report per period
6	Filling teacher performance questionnaire
7	Viewing student learning results assessment
8	Viewing the assessment results per class
9	Viewing student recapitulation result (transcript)
10	Viewing student achievement index
11	Viewing student cumulative grade index
12	Viewing student periodic achievement index
13	Managing teacher performance questionnaires
14	Managing the conversion of numeric to letter values

There are five users collaborate in this module; Pendidik, Kepala Pendidik, Perserta Didik, Tenaga Kepedidikan, and Tenaga Kepedidikan Pusat. The interaction and interplay of users to each feature can be describe in use case diagram as illustrated in Figure 4.5

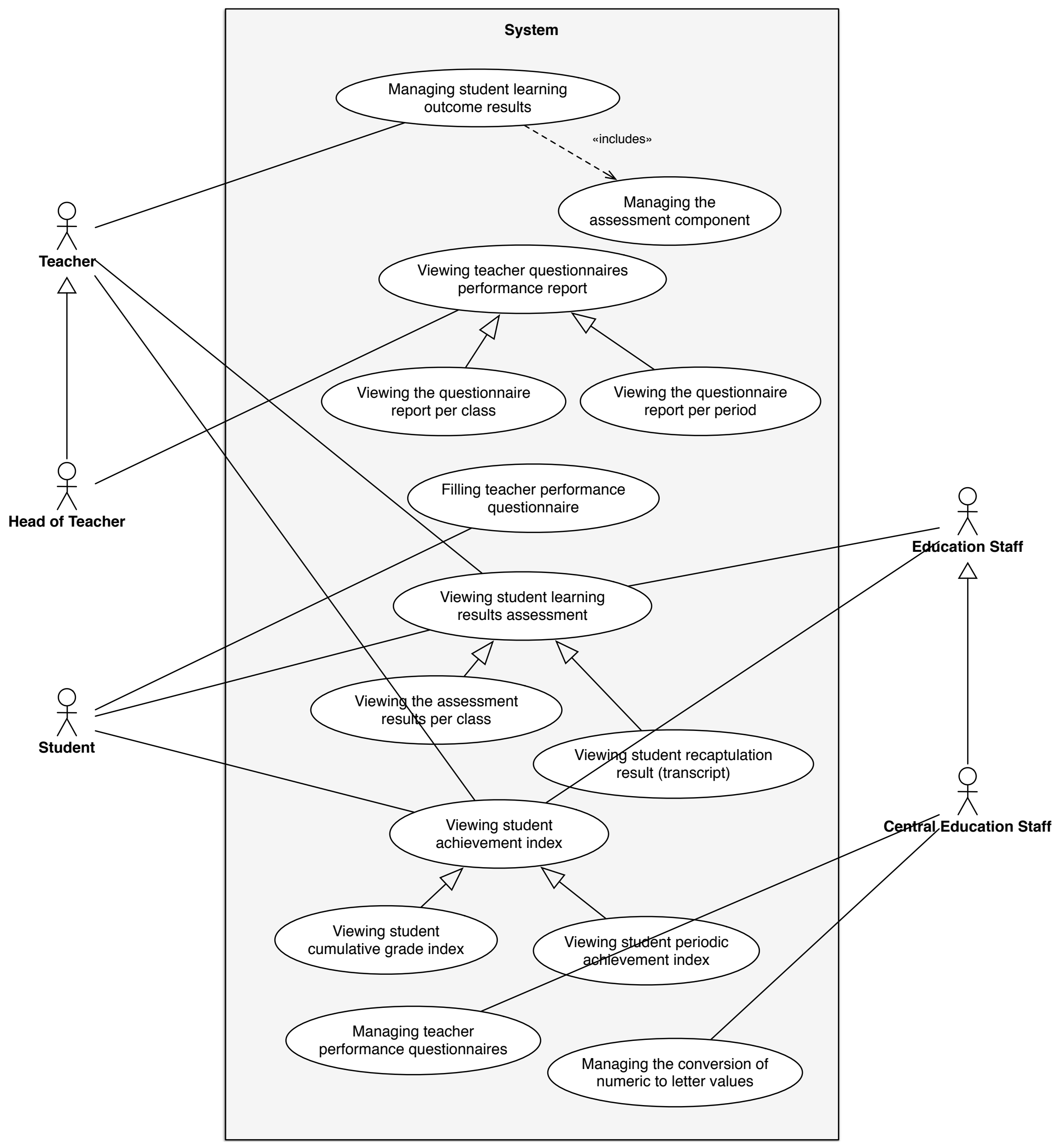

System
Managing student learning outcome results
«includes»
Managing the assessment component
Teacher
Viewing teacher questionnaires performance report
Viewing the questionnaire report per class
Viewing the questionnaire report per period
Head of Teacher
Filling teacher performance questionnaire
Education Staff
Viewing student learning results assessment
Student
Viewing the assessment results per class
Viewing student recaptulation result (transcript)
Viewing student achievement index
Central Education Staff
Viewing student cumulative grade index
Viewing student periodic achievement index
Managing teacher performance questionnaires
Managing the conversion of numeric to letter values

Figure 4.5 Use Case Diagram of SIA Grading Module

The grading module organizing *Java classes* and *interfaces* by categorized it in *packages* unique namespace, to represent parts and components of the system. The module consists of three main packages, there are *Package Controller, Package Service,* and *Package Repository.*

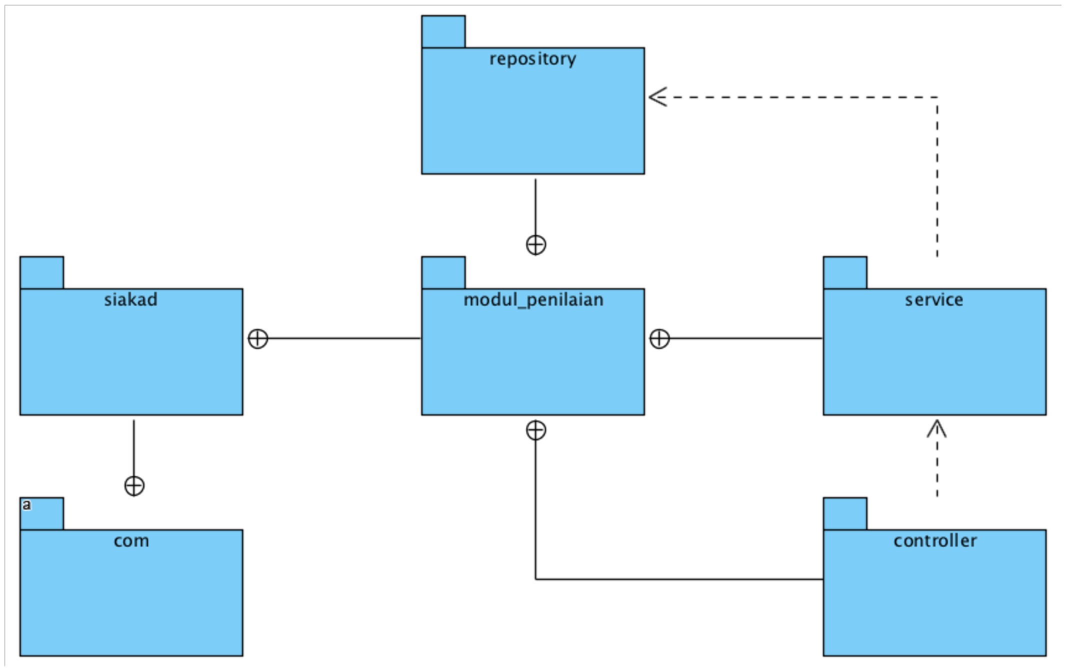


Figure 4.6 Package Diagram of Grading Module

The *Package Controller* (package com.siakad.modul_penilaian. controller), responsible for act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output.

The *Package Service* (package com.siakad.modul_penilaian.service), responsible for the middle layer between presentation and data store. It abstracts business logic and data access. It defines and implement the service interface and the data contracts

The *Package Repository* (package com.siakad.modul_penilaian. repository), responsible to separate the logic that retrieves the data and maps it to the entity model from the business logic that acts on the model. Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects. Package repository implemented Repository pattern in interacting with the database through Hibernate Framework as helper of Data Access Object (DAO).

The interconnection between packages of the module illustrated in package diagram in Figure 4.6.

Package Controller consist of 5 main classes. The class name, and its properties as show in Table 4.2 Package Repository consist of 18 classes and 18 interfaces. The class name, interfaces, and its properties as show in Table 4.3 Package Service consist of 24 classes and 18 interfaces. The classes name, interfaces, and its properties as show in Table 4.4

Table 4.2 Lists of classes and its properties in package controller

No.	**Class Name**	**Visibility**	**Parent Name**
1	ControllerFile	public	controller
2	ControllerIP	public	controller
3	ControllerKuisioner	public	controller
4	ControllerLaporan	public	controller
5	ControllerNilai	public	controller

Table 4.3 Lists of classes, interface, and its properties in package repository

No.	**Class Name**	**Stereotypes**	**Visibility**	**Parent Name**
1	IpkRepository	<<Interface>>	public	repository
2	IpkRepositoryImpl		public	repository
3	IpsRepository	<<Interface>>	public	repository
4	IpsRepositoryImpl		public	repository
5	KomponenNilaiRepository	<<Interface>>	public	repository
6	KomponenNilaiRepositoryImpl		public	repository
7	KonversiNilaiRepository	<<Interface>>	public	repository
8	KonversiNilaiRepositoryImpl		public	repository
9	KrsRepository	<<Interface>>	public	repository
10	KrsRepositoryImpl		public	repository
11	KuisionerRepository	<<Interface>>	public	repository
12	KuisionerRepositoryImpl		public	repository
13	MenuPeranRepository	<<Interface>>	public	repository
14	MenuPeranRepositoryImpl		public	repository
15	NilaiKuisionerRepository	<<Interface>>	public	repository

No.	Class Name	Stereotypes	Visibility	Parent Name
16	NilaiKuisionerRepositoryImpl		public	repository
17	NilaiRepository	<<Interface>>	public	repository
18	NilaiRepositoryImpl		public	repository
19	PdRepository	<<Interface>>	public	repository
20	PdRepositoryImpl		public	repository
21	PembRepository	<<Interface>>	public	repository
22	PembRepositoryImpl		public	repository
23	PendidikPengajarRepository	<<Interface>>	public	repository
24	PendidikPengajarRepositoryImpl		public	repository
25	PenggunaRepository	<<Interface>>	public	repository
26	PenggunaRepositoryImpl		public	repository
27	PeranPenggunaRepository	<<Interface>>	public	repository
28	PeranPenggunaRepositoryImpl		public	repository
29	PeranRepository	<<Interface>>	public	repository
30	PeranRepositoryImpl		public	repository
31	PertanyaanKuisionerRepository	<<Interface>>	public	repository
32	PertanyaanKuisionerRepositoryImpl		public	repository
33	StatusKuisionerRepository	<<Interface>>	public	repository
34	StatusKuisionerRepositoryImpl		public	repository
35	TglSmtRepository	<<Interface>>	public	repository
36	TglSmtRepositoryImpl		public	repository

Table 4.4 Lists of classes, interfaces, and its properties in package service

No.	Class Name	Stereotypes	Visibility	Parent Name
1	AjaxResponse		public	service
2	dataTranskrip		public	service
3	IpkService	<<Interface>>	public	service
4	IpkServiceImpl		public	service
5	IpsService	<<Interface>>	public	service
6	IpsServiceImpl		public	service
7	JSONNilai		public	service
8	JSONNilaiKuisioner		public	service
9	JSONPertanyaan		public	service

No.	Class Name	Stereotypes	Visibility	Parent Name
10	KomponenNilaiService	<<Interface>>	public	service
11	KomponenNilaiServiceImpl		public	service
12	KonversiNilaiService	<<Interface>>	public	service
13	KonversiNilaiServiceImpl		public	service
14	KrsService	<<Interface>>	public	service
15	KrsServiceImpl		public	service
16	KuisionerService	<<Interface>>	public	service
17	KuisionerServiceImpl		public	service
18	MenuPeranService	<<Interface>>	public	service
19	MenuPeranServiceImpl		public	service
20	NilaiKuisionerPerPemb		public	service
21	NilaiKuisionerService	<<Interface>>	public	service
22	NilaiKuisionerServiceImpl		public	service
23	NilaiService	<<Interface>>	public	service
24	NilaiServiceImpl		public	service
25	PdService	<<Interface>>	public	service
26	PdServiceImpl		public	service
27	PembService	<<Interface>>	public	service
28	PembServiceImpl		public	service
29	PendidikPengajarService	<<Interface>>	public	service
30	PendidikPengajarServiceImpl		public	service
31	PenggunaService	<<Interface>>	public	service
32	PenggunaServiceImpl		public	service
33	PeranPenggunaService	<<Interface>>	public	service
34	PeranPenggunaServiceImpl		public	service
35	PeranService	<<Interface>>	public	service
36	PeranServiceImpl		public	service
37	PertanyaanKuisionerService	<<Interface>>	public	service
38	PertanyaanKuisionerServiceImpl		public	service
39	StatusKuisionerService	<<Interface>>	public	service
40	StatusKuisionerServiceImpl		public	service
41	TglSmtService	<<Interface>>	public	service
42	TglSmtServiceImpl		public	service

The class design of the module follows the SOLID principles of object-oriented programming. As here, they implemented Dependency Inversion Principle. With Dependency Inversion Principle, an interface is introduced as an abstraction in a package. An object refers to interface and an object from another package inherits from interface too.

The Spring Framework use annotation in representing type of class. The @Service annotation is a stereotype and is used at class level that makes the class a service. A service class implements business logic using DAO, utility classes etc. The @Repository annotation is a stereotype and is used at class level. The class, whose behavior is to store, fetch or search data, comes to the repository category. The @Controller annotation is a stereotype and is used at class level in Spring Web MVC. It indicates that the class is a web controller.

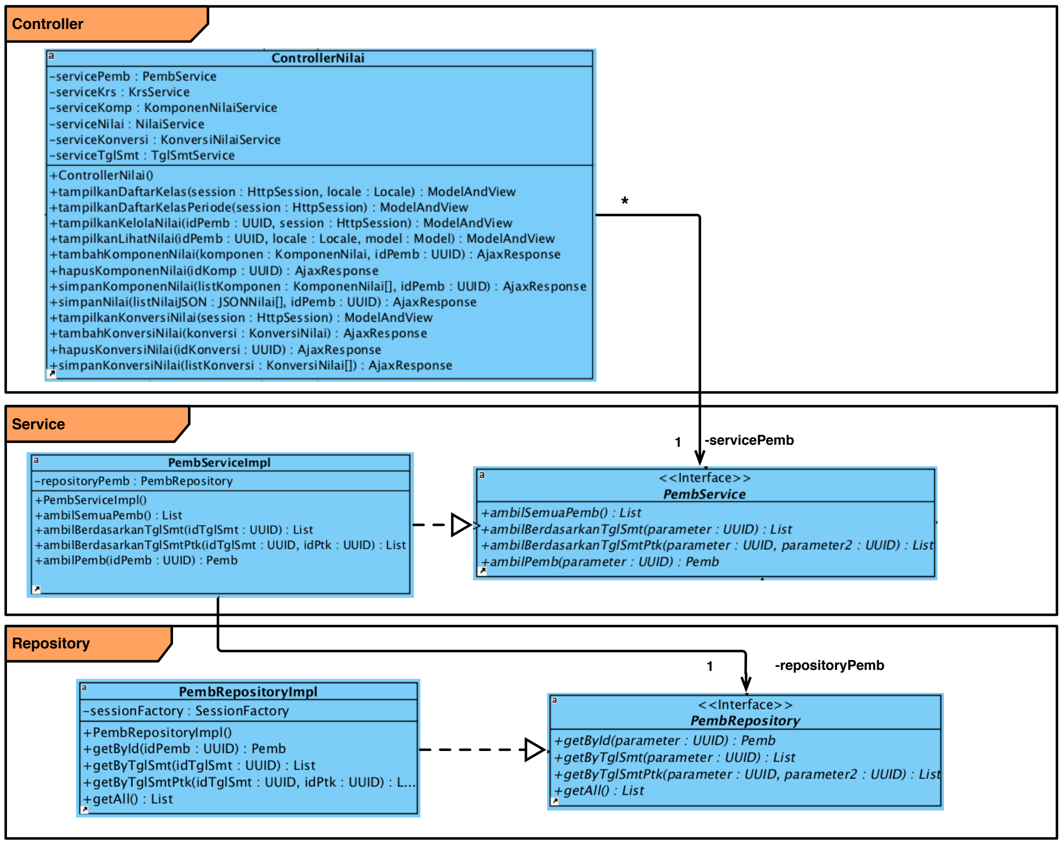

Controller
ControllerNilai
-servicePemb : PembService
-serviceKrs : KrsService
-serviceKomp : KomponenNilaiService
-serviceNilai : NilaiService
-serviceKonversi : KonversiNilaiService
-serviceTglSmt : TglSmtService
+ControllerNilai()
+tampilkanDaftarKelas(session : HttpSession, locale : Locale) : ModelAndView
+tampilkanDaftarKelasPeriode(session : HttpSession) : ModelAndView
+tampilkanKelolaNilai(idPemb : UUID, session : HttpSession) : ModelAndView
+tampilkanLihatNilai(idPemb : UUID, locale : Locale, model : Model) : ModelAndView
+tambahKomponenNilai(komponen : KomponenNilai, idPemb : UUID) : AjaxResponse
+hapusKomponenNilai(idKomp : UUID) : AjaxResponse
+simpanKomponenNilai(listKomponen : KomponenNilai[], idPemb : UUID) : AjaxResponse
+simpanNilai(listNilaiJSON : JSONNilai[], idPemb : UUID) : AjaxResponse
+tampilkanKonversiNilai(session : HttpSession) : ModelAndView
+tambahKonversiNilai(konversi : KonversiNilai) : AjaxResponse
+hapusKonversiNilai(idKonversi : UUID) : AjaxResponse
+simpanKonversiNilai(listKonversi : KonversiNilai[]) : AjaxResponse
*
Service
1
-servicePemb
PembServiceImpl
-repositoryPemb : PembRepository
+PembServiceImpl()
+ambilSemuaPemb() : List
+ambilBerdasarkanTglSmt(idTglSmt : UUID) : List
+ambilBerdasarkanTglSmtPtk(idTglSmt : UUID, idPtk : UUID) : List
+ambilPemb(idPemb : UUID) : Pemb
<<Interface>>
PembService
+ambilSemuaPemb() : List
+ambilBerdasarkanTglSmt(parameter : UUID) : List
+ambilBerdasarkanTglSmtPtk(parameter : UUID, parameter2 : UUID) : List
+ambilPemb(parameter : UUID) : Pemb
Repository
1
-repositoryPemb
PembRepositoryImpl
-sessionFactory : SessionFactory
+PembRepositoryImpl()
+getById(idPemb : UUID) : Pemb
+getByTglSmt(idTglSmt : UUID) : List
+getByTglSmtPtk(idTglSmt : UUID, idPtk : UUID) : L...
+getAll() : List
<<Interface>>
PembRepository
+getById(parameter : UUID) : Pemb
+getByTglSmt(parameter : UUID) : List
+getByTglSmtPtk(parameter : UUID, parameter2 : UUID) : List
+getAll() : List

Figure 4.7 ControllerNilai class diagram interconnection with other classes

For example, Class ControlNilai is one of controller class in package controller, it communicates to class PembService in package service and use class

PembRepository in package repository to connect and query require data from the database. Figure 4.7 demonstrate how ControllerNilai class diagram communicates and depends on other classes of another package.

The @Controller and @RequestMapping annotations allow flexible method names and signatures. In this example, the method accepts a Model and returns a view name as a String. @Controller and @RequestMapping and many other annotations form the basis for the Spring MVC implementation. The @Controller annotation indicates that a class serves the role of a controller. The @Controller annotation acts as a stereotype for the annotated class, indicating its role as shown in *ControllerNilai.java* class in package Controller. The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations. @RequestMapping annotation to map URLs such as /lihat_nilai onto an entire class or a handler method. Typically, the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

In the Table 4.5 shown a part of *ControllerNilai.java* example, @RequestMapping is used in a number of places. The first usage is on the type (class) level, which indicates that all handler methods in this controller are relative to the /lihat_nilai path. The post() method has a further @RequestMapping refinement: it only accepts POST requests, meaning that an HTTP POST for /lihat_nilai invokes this method.

A @Autowired annotation use to auto wire bean on the setter method, constructor or a field, and autowired property in a particular bean. The @Autowired here is autowired on properties to get rid of the setter methods.

Table 4.5 Source code of class ControllerNilai in package Controller

```
@Controller
public class ControllerNilai {
        @Autowired
        private PembService servicePemb;

@RequestMapping(value = "/lihat_nilai/", method = RequestMethod.POST)
```

```
	public ModelAndView tampilkanLihatNilai(@RequestParam("idPemb") UUID
idPemb, Locale locale, Model model) {
		List<TglSmt> daftarTglSmt = serviceTglSmt.ambilSemuaTglSmt();
		List<Pemb> kelas = servicePemb.ambilSemuaPemb();
		List<Krs> krsInfo = serviceKrs.ambilKrsBerdasarkanPemb(idPemb);
		Pemb pemb = servicePemb.ambilPemb(idPemb);
		String namaKelas = pemb.getMk().getNamaMK() + " " +
pemb.getNmPemb();

		ModelAndView lihatNilai = new ModelAndView();
		lihatNilai.setViewName("laporan_nilai_per_kelas");
		lihatNilai.addObject("krsInfo", krsInfo);
		lihatNilai.addObject("namaKelas", namaKelas);
		lihatNilai.addObject("listKelas", kelas);
		lihatNilai.addObject("listTglSmt", daftarTglSmt);

		return lihatNilai;
	}
}
```

@Component is a generic stereotype for any Spring-managed component. @Repository, and @Service are specializations of @Component or more specific use cases, for example in the persistence and service layers. In Table 4.6, and Table 4.7 are a part of *PembServiceImpl.java* and *PembRepositoryImpl.java* classes source code. In *PembServiceImp* class, @Service, act as business logic. This annotation of business layer in which our user will not directly call persistence method so it will call this method using @Service. @Service will request @Repository as per user request. The *PembRepositoryImpl* is a class is for persistence layer (Data Access Layer) of application which used to get data from database.

Table 4.6 Source code of class PembServiceImpl in package service

```
@Service
public class PembServiceImpl implements PembService {
	@Autowired
	private PembRepository repositoryPemb;

	@Override
	public List<Pemb> ambilSemuaPemb() {
		// TODO Auto-generated method stub
		return repositoryPemb.getAll();
	}

	@Override
	public List<Pemb> ambilBerdasarkanTglSmt(UUID idTglSmt) {
		// TODO Auto-generated method stub
		return repositoryPemb.getByTglSmt(idTglSmt);
	}
	@Override
	public List<Pemb> ambilBerdasarkanTglSmtPtk(UUID idTglSmt, UUID idPtk)
{
		// TODO Auto-generated method stub
		return repositoryPemb.getByTglSmtPtk(idTglSmt, idPtk);
	}

	@Override
	public Pemb ambilPemb(UUID idPemb) {
		// TODO Auto-generated method stub
		return repositoryPemb.getById(idPemb);
	}
}
```

Table 4.7 Source code of class PembRepositoryImpl in package repository

```
@Transactional
@Repository
public class PembRepositoryImpl implements PembRepository {
	@Autowired
	private SessionFactory sessionFactory;

	@Override
	public Pemb getById(UUID idPemb) {
		// TODO Auto-generated method stub
		return                                              (Pemb)
sessionFactory.getCurrentSession().get(Pemb.class, idPemb);
	}

	@Override
	public List<Pemb> getByTglSmt(UUID idTglSmt) {
		// TODO Auto-generated method stub
		Query                          query                     =
sessionFactory.getCurrentSession().createQuery("SELECT  pemb  FROM  Pemb  pemb
WHERE  pemb.tglSmt.idTglSmt  =  '"  +  idTglSmt  +  "'  AND  pemb.aPembTerhapus  =
FALSE");
		return query.list();
	}

	@Override
	public List<Pemb> getByTglSmtPtk(UUID idTglSmt, UUID idPtk) {
		// TODO Auto-generated method stub
		String queryString = "SELECT pp.pemb FROM PendidikPengajar pp
WHERE  pp.pemb.tglSmt.idTglSmt  =  '"  +  idTglSmt  +  "'  AND  pp.ptk.idPtk  =  '"  +
idPtk + "' AND pp.pemb.aPembTerhapus = FALSE";
		Query                          query                     =
sessionFactory.getCurrentSession().createQuery(queryString);
		return query.list();
	}

	@Override
	public List<Pemb> getAll() {
		Query                          query                     =
sessionFactory.getCurrentSession().createQuery("SELECT  pemb  FROM  Pemb  pemb
WHERE pemb.aPembTerhapus = FALSE");
		return query.list();
	}
}
```

4.1.2 Measuring Complexity of the Application

The following detail provide description gaining results complexity measurement of the case study.

Total number of packages (NOP) is 3 packages. Total number of classes (NOC) is 47 classes. Total number of interfaces (NOI) is 37 interfaces. Total number of attributes (NOF) is 79 attributes. Total number of methods (NOM) is 250 methods and total line of code (TLOC) represented in thousand-line of code (KLOC) is 3.117 KLOC.

The mean value of; McCabe Cyclomatic Complexity (MCC) is 1.268, the Weighted Methods per Class (WMC), sum of the McCabe Cyclomatic Complexity for all methods in a class is 6.745. The Lack of Cohesion of Methods (LCOM), a measure for the Cohesiveness of a class is 4.968. The Efferent Coupling (CE), the number of classes inside a package that depend on classes outside the package is 25.667. The Afferent Coupling (CA), the number of classes outside a package that depend on classes inside the package is 7.667.

The graph of the complexity result of the grading module presented in Figure 4.8 and Figure 4.9.

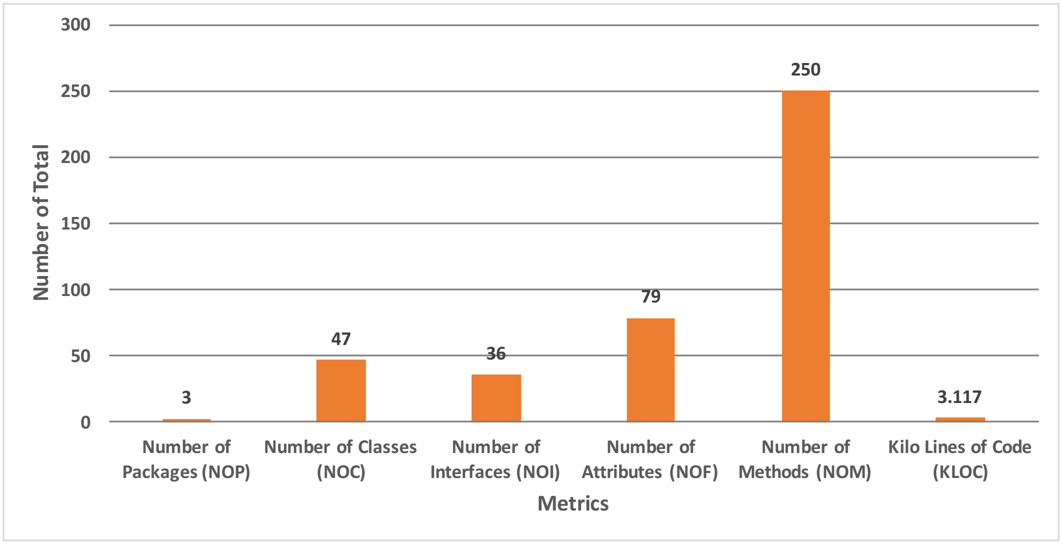


Figure 4.8 Graph representation number of total of metrics

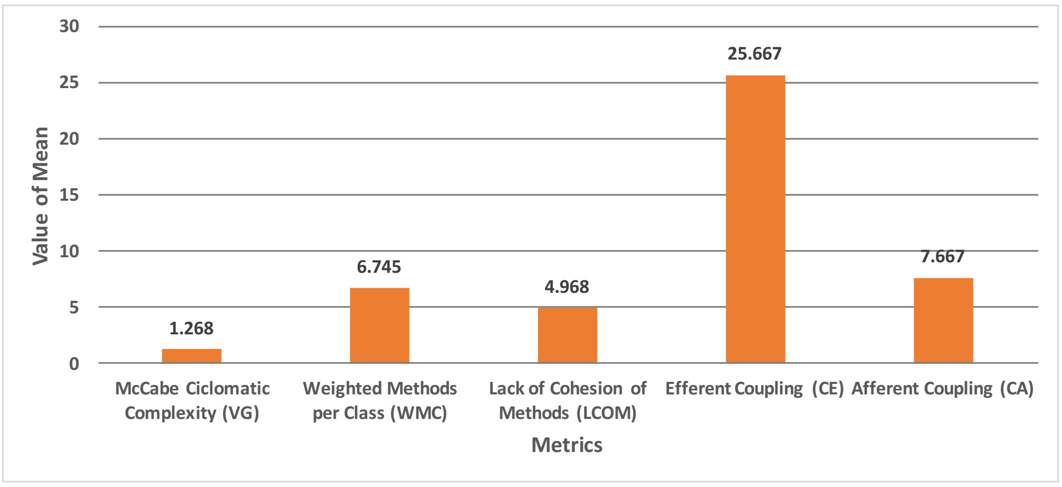


Figure 4.9 Graph representation of mean metric value of complexity

4.1.3 Identifying Problem

There is architecture and design patterns applied on the Grading module. The SIA system is created based on Hierarchical model–view–controller (HMVC), software architectural pattern as main pattern of its structure and implementing single responsibility principle. The system is divided into layers follow the layering principle. Layering principle consist of Presentation Layer, Service Layer (the actual business logic) and Data Access Layer. The system source code structure and design is powered and implemented by several technology and framework as Java EE platform, Spring Framework and Hibernate. And the system is deployed on Virgo server which Apache Tomcat version.

In Grading module, a Controller is typically responsible for preparing a model Map with data and selecting a view name but it can also write directly to the response stream and complete the request. View name resolution is highly configurable through file extension or accept header content type negotiation, through bean names, a properties ViewResolver file. The model (the M in MVC) is a map interface, which allows for the complete abstraction of the view technology. It can integrate directly with template based rendering technologies as JSP. The model map is simply transformed into an appropriate format inform of JSP request attributes and rendering to user web browser as result of complete request and response.

In Figure 4.10 classes and subclasses, especially older ones are masses of complex legacy code. When class in package controller must interact, they often make calls directly into classes package services. It is creating one-to-many dependencies, and these myriad tendrils of connectivity are difficult to maintenance. The subclasses become very delicate since making seemingly insignificant changes in a single subclass can affect the entire program. It creates complexity communication and dependencies between two or more classes or interfaces.

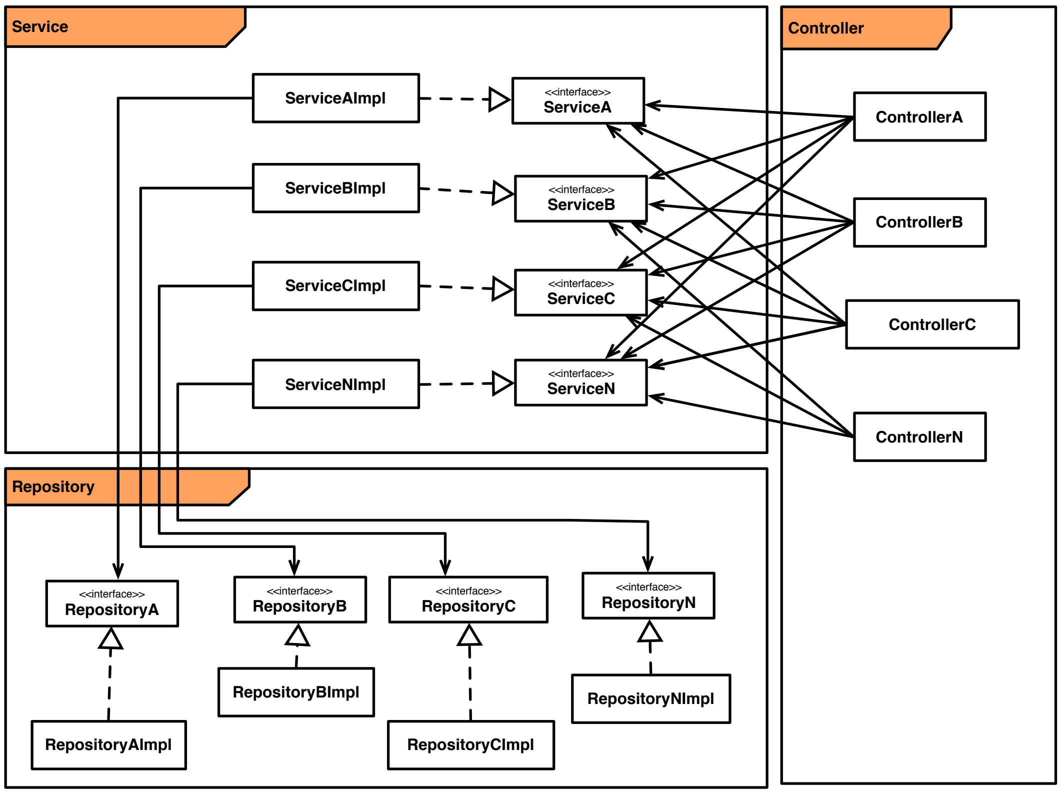


Figure 4.10 Communication and dependencies between classes

For example, class ControllerA in package controller, make communication with four interfaces of package service, ServiceA and ServiceB, ServiceC and ServiceN. Class ControllerB and ControllerC and ControllerN also create communication to ServiceA and ServiceB, ServiceC and ServiceN too. It means all *Services* class have to handle three or more communications at the same time, it is created dependency nightmare for developer in future maintenance. By this potential emerging problem, we consider to redesign and refactor the module system structure for further feature extend and performance of the system.

4.1.4 Design Patterns Selection

According to problem identification section, we chose to introduce the advantages of apply Façade pattern in decreasing dependency for the purpose of separation of concerns (SoC).

MVC pattern, especially in enterprise system and complex code, are masses of dependency code and poorly designed, over-complex classes. When two classes of packages must interact, they often make calls directly into each other, and these create large connectivity maintenance nightmare. The classes become very delicate since making seemingly insignificant changes in a single classes can affect the entire program. Facade addresses the problem by forcing programmers to use a 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. Facade improves the independence of the classes, making it easy to change or even replace them without impacting outside code.

Façade improves the independence of the subsystems (classes), making it easy to change or even replace them without impacting outside code. It provides a manageable way to migrate legacy code to a more object-oriented structure, hides badly done, overly complex legacy code and lets you treat an entire legacy system as if it were a single, coarse-grained object. It deals internally with all the actions you would otherwise have to code each time you access those functionalities; the result is simplification of calls to an action on the class.

The Facade pattern ensures the scalability of the Models. When a number of Model objects grows, and it becomes inconvenient to pass a lot of them to Controllers, or when you notice a lot of business logic settling down in Controllers, consider wrapping those Model objects in a Facade which will hide the complexity and implementation details. In our case study, the Façade is a class that provide interface access to an object from the container, from Controllers to Models or services. The Façade wraps all service classes and create interface ease of use for controller to reduce complexity dependency and make certain the scalability between those classes.

4.2 Refactoring

4.2.1 Refactoring and Applying Design Patterns

In Refactoring processes, we identify refactoring candidates, plan refactoring activities, implement on planned refactoring tasks, and test to ensure behavior preservation.

We identify the refactoring candidates by introduce to advance Façade design pattern. The create façade service package that contain façade class function that use for functioning easy to use interface communication between classes in package service and package controller.

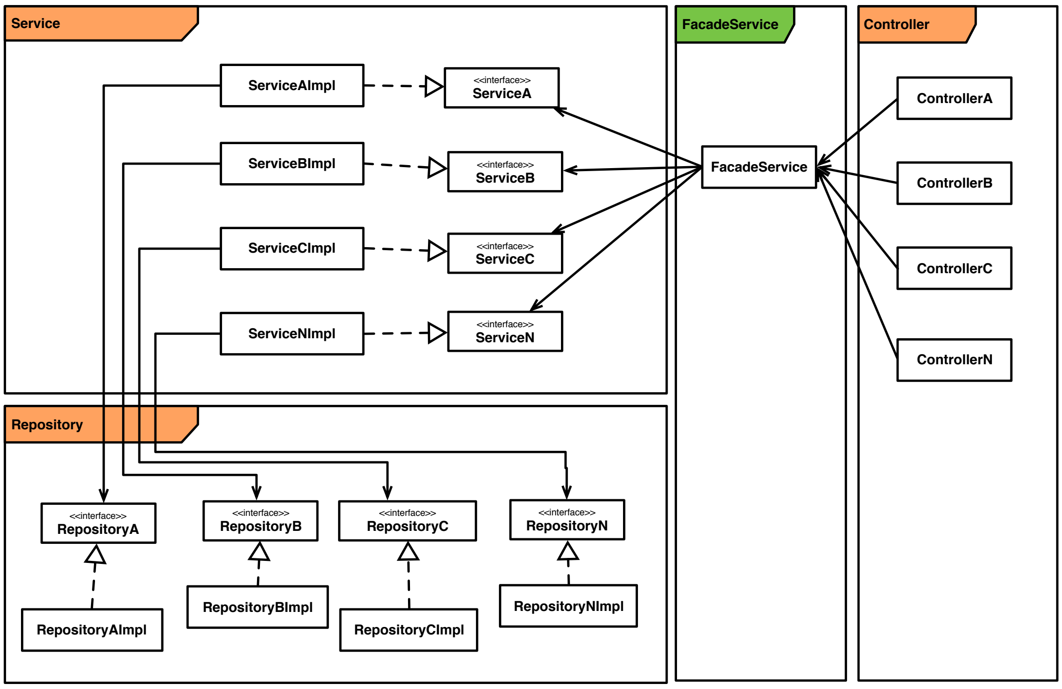


Figure 4.11 Class diagram of the Façade pattern implement in grading module

In Figure 4.11 present our candidate façade design pattern design in grading module. FacadeService is placed in between the controller and the service. It created opportunities to establish intermediate layers of abstraction with wrap a poorly-designed collection of classes with a single well-designed classes that further foster reduced levels of coupling and reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system. This allows the service to remain decoupled from the controller. A façade component is used to abstract a part of the service and controller architecture with less-coupling potential

This solution is to attain a reduced degree of coupling between services and controller, thereby increasing the freedom and flexibility with which services can be individually evolved. This can result in an elegant architecture design with clean layers of abstraction, but it can also impose extra processing overhead that naturally comes with increasing the physical distribution of controller call.

4.3 Performance Measuring

4.3.1 Measuring Complexity of the Applications

The finding complexity result of the application variants can be identified. The complexity result can be grouping into two categories, number of total and complexity metric.

Figure 4.12 illustrates the number of total both legacy and refactored SIA. Total number of packages (NOP) is 3 packages for legacy and 3 packages for refactored. Total number of classes (NOC) is 47 classes for legacy and 48 classes for refactored. Total number of interfaces (NOI) is 36 interfaces for legacy and refactored. Total number of attributes (NOF) is 79 attributes for legacy and 82 attributes for refactored. Total number of methods (NOM) is 250 methods for legacy and 261 methods for refactored. And total line of code (TLOC) represented in thousand-line of code (KLOC) is 3.117 KLOC for legacy and 3.427 KLOC for refactored.

In complexity metric, the result gained from the experiment and presented in Mean value standard. Figure 4.13 show the mean value results of complexity metric both legacy and refactored of the grading module.

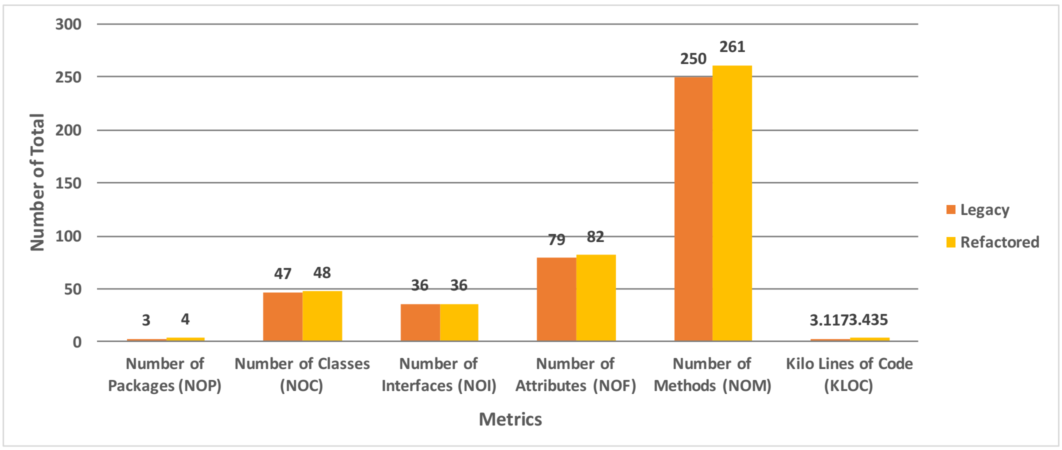


Figure 4.12 Graph representation number of total of metrics of legacy and refactored system

The mean value of McCabe Cyclomatic Complexity (MCC) is 1.268 for legacy and 1.598 for refactored. The mean of Weighted Methods per Class (WMC), is 6.745 for legacy and 7.253 for refactored. The mean of Lack of Cohesion of Methods (LCOM) is 4.968 for legacy and 3.891 for refactored. The mean value of Efferent Coupling (CE) is 25.667 for legacy and 19.891 for refactored. The mean value of Afferent Coupling (CA) is 7.667 for legacy and 5.376 for refactored.

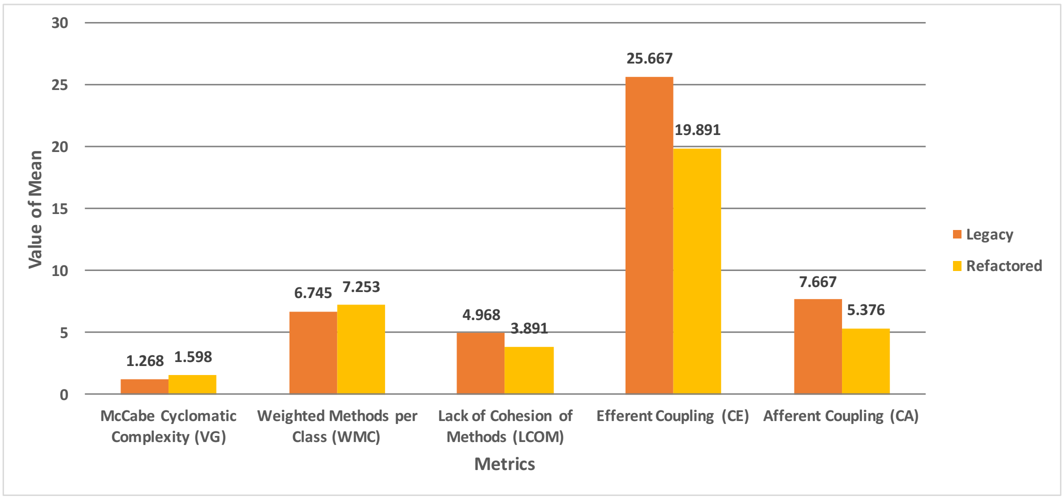


Figure 4.13 Graph representation of mean value complexity of legacy and refactored system

The comparison of the complexity result shown clear differences perspective of metric between legacy and refactored.

There is the same result in Total number of packages (NOP). Total number of classes (NOC) is increased 1 class. Total number of interfaces (NOI) is still the same. Total number of attributes (NOF) is increased 3 attributes. Total number of methods (NOM) is increased 11 methods. Total line of code (TLOC) represented in line of code (LOC) is increased 310 LOC. The result is increased because of we added package and class in investigate new design of the refactored system.

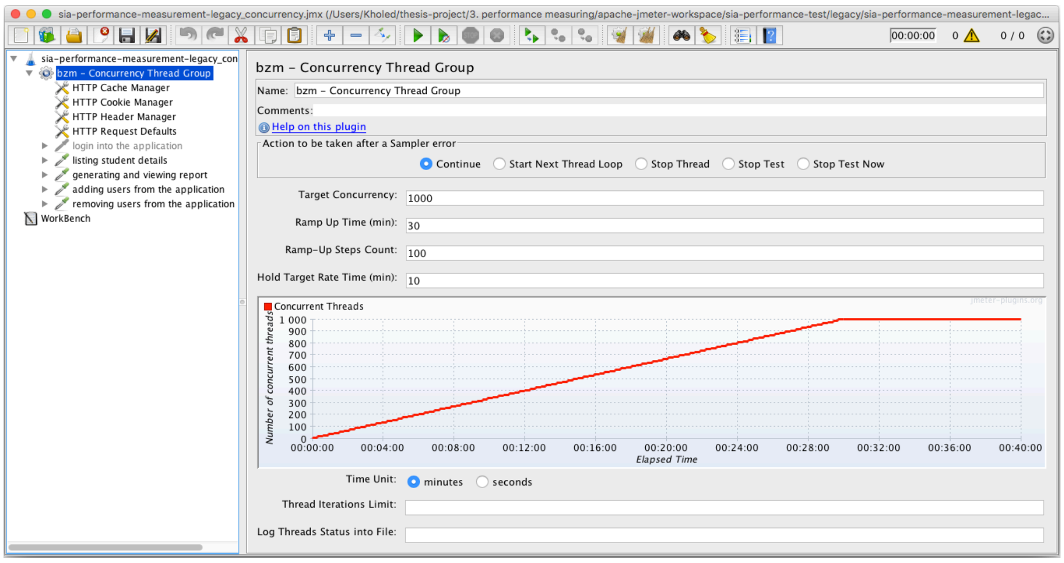
The mean value of McCabe Cyclomatic Complexity (VG) is increased 0.330 point. The mean of Weighted Methods per Class (WMC), sum of the McCabe Cyclomatic Complexity for all methods in a class is increased 0.508 point. The mean of Lack of Cohesion of Methods (LCOM), a measure for the Cohesiveness of a class is decreased 1.077 point. The mean value of Efferent Coupling (CE), the number of classes inside a package that depend on classes outside the package is decreased 5.776 point. The mean value of Afferent Coupling (CA), the number of classes outside a package that depend on classes inside the package is decreased 2.291 point. Complexity metric as MCC and WMC is calculated based on function point of method so the result of Mean value is almost increased in this class because we added methods of class. Adding new extra layer of façade design pattern, the mean value of metrics LCOM, CE, and CA is decreased because of these metrics related to dependency between two object inside or outside object.

4.3.2 Dynamic Performance Efficiency Measurement

There are four test scenarios in the test includes activities typical for the application. There are Test Scenario 1 (TS1): listing student details, Test Scenario 2 (TS2): generating and viewing report, and Test Scenario (TS3): adding users the application and Test Scenario 1 (TS4): removing users from the application.

All the test scenarios were simulated with a number of concurrent threads (users) increasing from 10 to 350. This final number of threads was determined empirically and it was the maximum number of threads that the application and server could handle (breaking point). After running a full sequence of requests for given number of threads, it was repeated until the total number of requests reached around 3,150 requests. This number was also determined empirically and it was when the response time from the server was stable, meaning that the server had already allocated enough resources to serve a given number of threads. Table 4.8 show the JMeter log file determined the application server stable response the request.

A test round for one tested case started from simulating 10 concurrent threads. Then the number was set to 10 threads and after that it was always increased by 10 until the maximum number of 350 threads was reached. Each round was repeated 3 times to ensure that the results are meaningful and reliable.



sia-performance-measurement-legacy_concurrency.jmx (/Users/Kholed/thesis-project/3. performance measuring/apache-jmeter-workspace/sia-performance-test/legacy/sia-performance-measurement-legac...

- sia-performance-measurement-legacy_con
  - bzm - Concurrency Thread Group
    - HTTP Cache Manager
    - HTTP Cookie Manager
    - HTTP Header Manager
    - HTTP Request Defaults
    - login into the application
    - listing student details
    - generating and viewing report
    - adding users from the application
    - removing users from the application
- WorkBench

bzm - Concurrency Thread Group

Name: bzm - Concurrency Thread Group

Comments:

Help on this plugin

Action to be taken after a Sampler error: [✓ Continue] Start Next Thread Loop / Stop Thread / Stop Test / Stop Test Now

Target Concurrency: 1000

Ramp Up Time (min): 30

Ramp-Up Steps Count: 100

Hold Target Rate Time (min): 10

Time Unit: [✓ minutes] seconds

Thread Iterations Limit:

Log Threads Status into File:

Figure 4.14 JMeter Concurrency Thread Group setting

We defined the Thread Group for pool of users that will execute a particular test case against the server. JMeter makes the number of users, and the ramp-rate configurable. We use HTTP Request Defaults configuration element to the Thread Group. This configuration element sets up the domain IP address of the server, the port and the protocol (HTTP/ HTTPS). We use HTTP Cookie Manager, it stores and sends cookies. HTTP Request and the response contains a cookie, the Cookie Manager automatically stores that cookie and will use it for all future requests. For the purposes of this research, the default configurations are enough. We define HTTP Header Manager, it lets you add or override HTTP request headers. The HTTP Cache Manager is used to add caching functionality to HTTP requests within its scope to simulate browser cache feature. Each Virtual User thread has its own Cache. By default, Cache Manager will store up to 5000 items in cache per Virtual User thread. We use HTTP Request element, for send an HTTP/HTTPS request to the SIA web server. This configuration element lets us sets up test scenarios as defined, the domain or IP address patch of web application.

In Figure 4.14, it is show concurrency thread group setup in JMeter. We set 1,000 threads as target load, 30 minutes Ramp Up Time, 100 Ramp-Up Steps, 10 minutes holding the target rate. This means that, the test begins immediately when JMeter starts. In every 0.3 minutes 10 users will be added until we reach 1000 users. It is can be calculate as 30 minutes divided by 10 steps equals 0.3 minutes per step. 1000 users divided by 100 steps equals 10 users per step. Totaling 10 users every 3 minutes. The first step is 0-10, the second 11-20, and 21-30 etc., because it started 10 threads to run at the beginning. After reaching 1,000 threads all of them will continue running and hitting the server together for 10 minutes and all thread will stop.

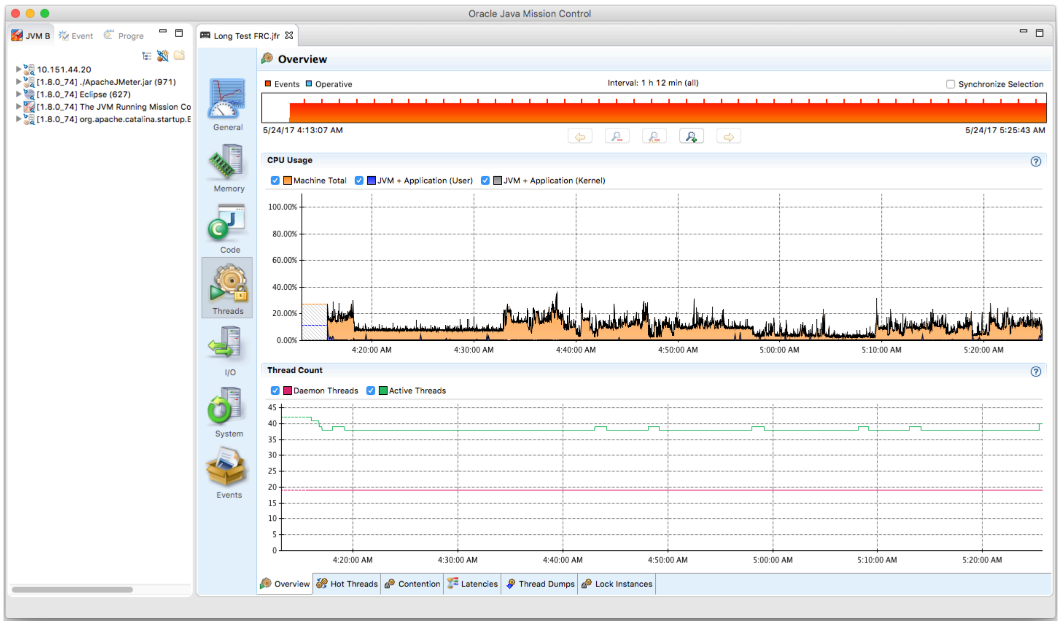


Figure 4.15 Java Flight Recorder (JFR) in Java Mission Control (JMC)

Java Mission Control (JMC) provide us to gather the data necessary with the lowest possible impact on the running system. JMC use the JMX Console as tool for monitoring and managing multiple Oracle JDK instances. It captures and presens live data about memory and CPU usage. Java Flight Recorder and Java Mission Control together create a complete tool chain to continuously collect low level and detailed runtime information enabling after-the-fact incident analysis.

In Figure 4.15 shown how we setup and captured data using Java Flight Recorder in Java Mission Control (JMC). We connect JMC profiling tool to SIA server through JMX connection. We use Java Flight Recorder (JFR) in produces detailed recordings about the JVM and the application it is running. The recorded memory and CPU usage data can be analyzed off line, using the Flight Recorder tool in JMC. We use two provides specialized tabs in JFR that focus on a specific area of Memory and Threads (CPU).

The result of dynamic performance efficiency measuring is provided variants depending on type of performance efficiency parameter measures and test scenarios. Each detail result provided as below.

4.3.3 Time Behavior Measures

1. Mean response time

The mean response times for the SIA application for legacy and refactored are shown in Figure 4.17. The mean response time chart shows how the differences between the implementations increased while the number of simulated users increased.

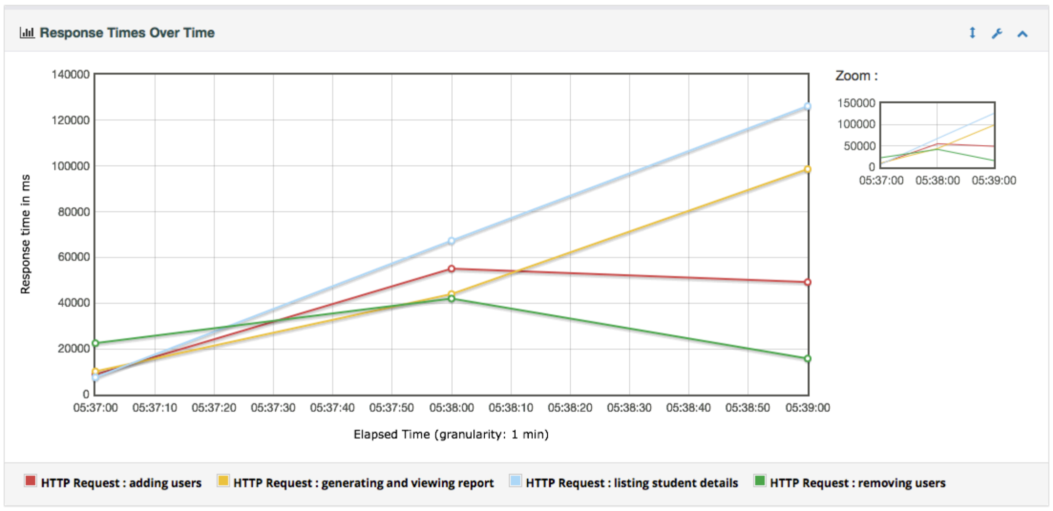


Figure 4.16 Graph Response time

Table 4.8 JMeter log output file

Sample	Start Time	Thread Name	Sample Time (ms)	Status	Bytes	Sent Byte	Latency	Connection Time (ms)
1	37:03.7	sia - penilaian 1-2	630	Success	118701	186	338	4
2	37:03.7	sia - penilaian 1-1	731	Success	118701	186	357	4
3	37:03.7	sia - penilaian 1-4	1062	Success	118701	186	360	3
9	37:03.7	sia - penilaian 1-6	1326	Success	118701	186	405	2
10	37:03.8	sia - penilaian 1-9	1316	Success	118701	186	388	2
22	37:03.8	sia - penilaian 1-19	1862	Success	118701	186	1587	4
29	37:03.9	sia - penilaian 1-29	2081	Success	118701	186	1861	2
30	37:03.9	sia - penilaian 1-30	2098	Success	118701	186	1894	3
37	37:03.9	sia - penilaian 1-37	2443	Success	118701	186	2223	4
40	37:04.0	sia - penilaian 1-41	2516	Success	118701	186	2264	6
49	37:04.0	sia - penilaian 1-49	2684	Success	118701	186	2627	4
50	37:04.0	sia - penilaian 1-51	2810	Success	118701	186	2749	7
59	37:04.1	sia - penilaian 1-60	3079	Success	118701	186	2909	8
60	37:04.0	sia - penilaian 1-55	3120	Success	118701	186	2946	4
70	37:04.1	sia - penilaian 1-70	3331	Success	118701	186	3141	73
80	37:04.2	sia - penilaian 1-79	3789	Success	118701	186	3718	101
90	37:04.2	sia - penilaian 1-89	4090	Success	118701	186	3884	140
99	37:04.3	sia - penilaian 1-96	4320	Success	118701	186	3972	122
100	37:04.3	sia - penilaian 1-102	4290	Success	118701	186	3944	154
150	37:04.7	sia - penilaian 1-163	6089	Success	118701	186	6035	177
190	37:05.0	sia - penilaian 1-218	7641	Success	118701	186	7497	1195
200	37:05.4	sia - penilaian 1-293	7944	Success	118701	186	7845	1196
250	37:04.8	sia - penilaian 1-190	11742	Success	118701	186	11530	4827
270	37:04.9	sia - penilaian 1-194	13056	Success	118701	186	12743	4818
290	37:05.4	sia - penilaian 1-283	14764	Success	118701	186	14475	5977
300	37:05.2	sia - penilaian 1-245	17105	Success	118701	186	16697	4837
323	37:05.3	sia - penilaian 1-275	125320	Warning	6866	3171	17474	12988
324	37:05.6	sia - penilaian 1-313	125138	Warning	4866	1579	23301	22282
327	37:05.5	sia - penilaian 1-311	125601	Success	18365	3171	24271	22278
328	37:05.8	sia - penilaian 1-350	125425	Warning	6616	2972	16485	8311
329	37:05.5	sia - penilaian 1-305	125693	Warning	5866	2375	23951	22278
340	37:05.5	sia - penilaian 1-302	126129	Warning	6366	2773	18553	12985
349	37:05.3	sia - penilaian 1-276	129251	Success	19115	3768	17777	12987
350	37:05.5	sia - penilaian 1-300	137474	Success	19115	3768	18678	12988

However, the character of the change was linear and proportional for each measurement point. It is also clearly visible that the mean response time rapidly decreased (require much time) when the number of simulated users exceeded about 350 users, which was the breaking point of the SIA application where the server could not handle the increased load. After that point the average response time values decreased to remain at an almost constant level until the end of the measured range. The reason for this is made clear after analyzing the success rate results.

It is clearly visible that Facade had the longest response time throughout the whole measurement. The legacy SIA had an average response time which was about 1.1096 conformance less than for the refactored SIA variant. The differences were visible even after reaching the breaking point of the server.

In Table 4.9 show the different each round and mean result in numeric of the two systems. There are three rounds of the test. The result present in each round of the test by calculate its mean value. For each test case can be summary in a mean value by calculate all round. The response time measure in millisecond (ms) unit.

Table 4.9 Mean response time

Round No. / metric	**Legacy**				**Refactored**			
	TS1	**TS2**	**TS3**	**TS4**	**TS1**	**TS2**	**TS3**	**TS4**
Round1	31685	61700	42681	17932	31885	80710	51252	18642
Round2	29264	64114	41065	14970	28926	67141	45863	19232
Round3	17894	56049	44685	16026	16945	58079	50281	17106
Mean	**26281**	**60621**	**42810**	**16309**	**25919**	**68643**	**49132**	**18327**

2. Response time conformance

The result of response time conformance measure is provided variants. In Table 4.10 show the result of comparing between legacy and refactored system. The result show that the refactored system takes much time in response the requests. Response time conformance usually smaller is better and less that 1 is good. The compared test results and test scenarios show that, refactored system has negative impact in response time conformance defectively, except in test scenario 1 (TS1).

Table 4.10 Response time conformance

System / metric	Test Scenario			
	TS1	**TS2**	**TS3**	**TS4**
Legacy	26281	60621	42810	16309
Refactored	25919	68643	49132	18327
Conformance	**0.9862**	**1.1323**	**1.1477**	**1.1237**

3. Throughput conformance

Throughput here is calculated as requests/sec unit of time. The time is calculated from the start of the first sample to the end of the last sample. This includes any intervals between samples, as it is supposed to represent the load on the server. The basic formula is: Throughput = (number of requests) / (total time).

The throughput results also showed clear differences between the investigated design patterns. This is shown in Table 4.11, result values smaller is better and the default best value is 0. The throughput values remained at an almost constant level until the servers breaking point, and the differences between the implementations also remained proportional. Similar to the results for average response times, the throughput values increased after the simulation passed the breaking point.

Table 4.11 Throughput conformance

System / metric	Test Scenarios				
	TS1	**TS2**	**TS3**	**TS4**	**Total**
Legacy	2.5/sec	2.4/sec	2.5/sec	2.6/sec	10.0/sec
Refactored	2.5/sec	2.3/sec	2.3/sec	2.5/sec	9.6/sec
Conformance	**1.000**	**1.043**	**1.087**	**1.040**	**1.042**

4.3.4 Resource Utilization Measures

1. Mean Processor utilization

Here we use CPU utilization to investigate performance, amount of time and percentage of used and CPU use for process a given task. It can be used to track

CPU performance regressions or improvements, and is a useful data point for performance problem investigations.

There is a little different in CPU usage between legacy and refactored SIA system. In Figure 4.17 and Figure 4.18 are graphs comparison CPU usage of both systems. Its result shown average CPU usage is 15% for the legacy and 17% for the refactored SIA system. The hot thread are the Main function 37.88% and 39.61% CPU usage and the Local Descriptor Scanner function 16.16% and 16.62% CPU usage for both systems.

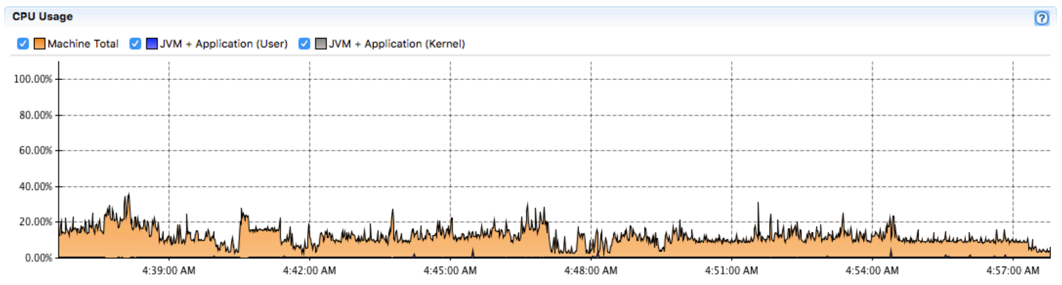


Figure 4.17 CPU usage on legacy system

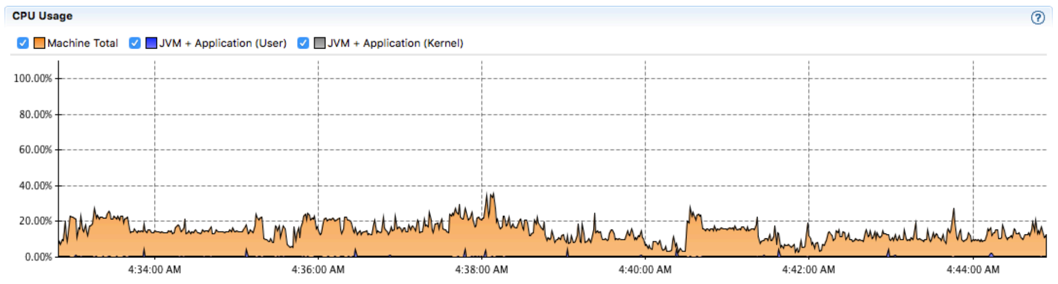


Figure 4.18 CPU usage on refactored system

2. Mean memory utilization

Memory utilization function on Java Mission Control allow us to check the memory allocation rate in our running application.

As in Figure 4.20 and Figure 4.21 show the memory usage for stored temporary data in executing the given tasks. The finding result show that refactored system used Memory Allocated for TLABs (Thread Local Allocation Buffer) is 574.79 Mega Bytes greater than legacy system that is 572.98 Mega Byte.

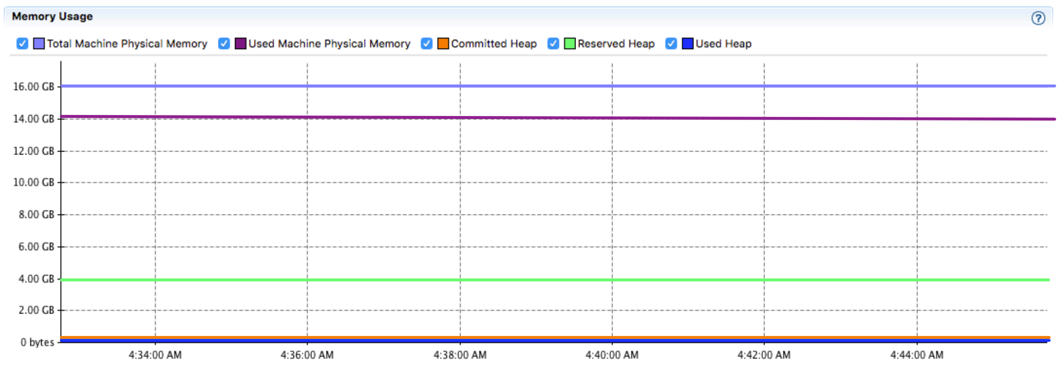


Figure 4.19 Memory usage on legacy system

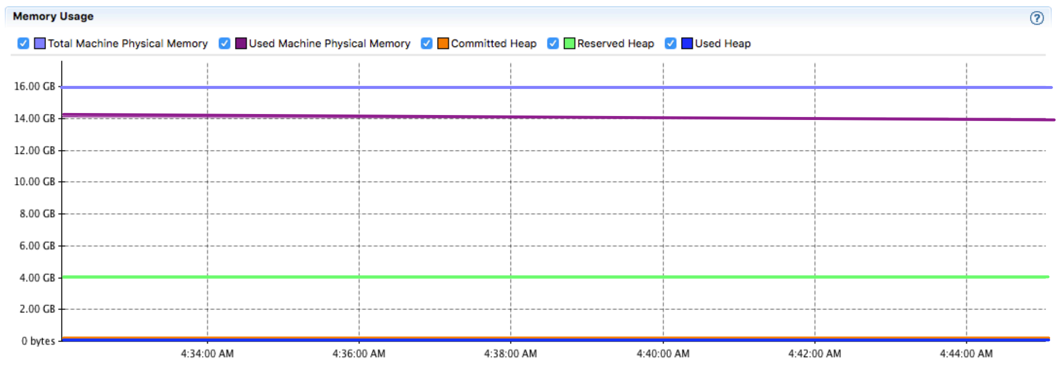


Figure 4.20 Memory usage on refactored system

4.3.5 Capacity Measures

1. Transaction processing capacity conformance

The percentage of error reflex the successful requests is depicted in Table 4.8 and Table 4.9 for conformance. Based on the results obtained for the percentage of successful and error responses, the measurement point for 350 users was identified as the point where the server could not handle the increased load and the requests resulted in errors. Since the failed requests were not processed entirely, their handling times were shorter compared to the handling times of fully-processed requests. The average response times lowered when the number of simulated users passed the breaking point that the application fully serve.

Table 4.12 Transaction processing error rate

Round No.	Test Scenarios / Legacy				Refactored			
	TS1	**TS2**	**TS3**	**TS4**	**TS1**	**TS2**	**TS3**	**TS4**
Round1	9.86%	41.05%	26.91%	8.45%	7.37%	24.40%	34.27%	6.55%
Round2	10.18%	43.00%	27.02%	7.93%	8.29%	37.53%	16.54%	8.57%
Round3	6.86%	38.86%	26.86%	6.57%	7.23%	28.48%	22.53%	5.69%
Mean	**8.97%**	**40.97%**	**26.93%**	**7.65%**	**7.63%**	**30.14%**	**24.45%**	**6.94%**

Table 4.13 Transaction processing error rate conformance

	Test Scenarios				
	TS1	**TS2**	**TS3**	**TS4**	**Total**
Legacy	8.97%	40.97%	26.93%	7.65%	21.13%
Refactored	7.63%	30.14%	24.45%	6.94%	17.29%
Conformance	**0.85**	**0.736**	**0.91**	**0.91**	**0.82**

4.3.6 Output Analysis and Validation

Based on the results for the legacy SIA and refactored solutions, especially by looking at the legacy results compared to the refactored results, it is possible to indicate the main reasons for the overall performance differences in the investigated configuration variants. In the case of the legacy SIA, there is provide default result. However, in the refactored SIA, differences were clearly noticeable. The Façade pattern, provide positive and negative impact in refactoring. It is by default create more heap of response time but it provides nice structure of code implement and maintenance.

The average response time is increasing almost linearly in all the deployment cases along with increasing number of clients. This is illustrated in Figure 4.16. The throughput of the application with Facade pattern deployed is increasing similar in cases of mean response time. The façade provided difference between the legacy, it is a more steeply decreasing throughput conformance. Additionally, refactored applications have much flatter characteristic of throughput with respect to the increasing number of requests. The rate of transaction processing successful requests has similar characteristics for all the cases. In the beginning it is 21% error in average for all test scenarios, while witch refactored system it is decrease to 17% error.

(This page intentionally left blank)

CHAPTER 5
CONCLUSION

In this chapter, we summarize and concludes the findings and contribution of the thesis and of our experimental evaluation. In addition, we provide remark and an outlook on possible future work.

5.1 Conclusions

This thesis describes an enterprise software system in which the design patterns is applied. The thesis begins with the important of performance efficiency of a system application, utilized design patterns in refactoring the legacy system, the academic information system in Indonesia, goes through the design, analyze and refactoring process, ends at the performance efficiency measuring, and analyze and evaluate the result.

Within the thesis, the SIA was analyzed and studied through several tool and reverse engineering technique. The Penilaian (grading) module was select to study as the main case implementation.

As theoretical parts, software designs and patterns are introduced in brief; the Java EE, as an industrial standardization, was introduced from its architecture and technology perspectives. Based on the theoretical parts, the SIA application was studied at a high and low level. The system architecture and its alternative were first reverse engineering. The current architecture obeyed the HMVC architecture and applied Java EE via Spring Web MVC framework. The database stored system is PostgreSQL, the hibernate framework as libraries helped in build the connection. The application was divided into six main modules. The grading module was architected in three main layers, presentation, business and service layer. After the studied of the SIA architecture, Façade design patterns were used to elaborate the architectural design and to provide the proven solutions to the recurring problems in the context of SIA realization. More design patterns are even being created and documented by software developers. The Java EE software developers, in real life, have their own options to use the different design patterns to solve their special problems in the particular context.

With the development of enterprise software, the design patterns will be much widely applied and adapted to solve recurring problem even in enterprise software. This work covers with three type of performance efficiency measures. There are time behavior, resource utilization, and capacity measures. Each measurement function provides a unique parameters and different varieties kind of result respect to performance of software quality.

The tests carried out 4 test scenarios with respect to the system and implemented design pattern, SIA and Façade design pattern. All the data gathered shows differences between the compared legacy and refactored of it implement design pattern, in terms of performance efficiency related quality factors of the application. In all the presented tests, the refactored implementation that used Facade design pattern had the highest response time and throughput. In addition, for mean result of the test scenario the characteristics of success rate 4 percentage were different. The number of users when the application stopped handling requests properly tends to be the highest also for refactored SIA application implementation. Additionally, the presented differences between the legacy and refactored system indicate consequences of particular implementation choices. The refactored system increased the number of network calls, which resulted in poor performance than the legacy system. Facade was clearly better in managing design of the system but worse in reduced response time and throughput especially when implemented in system that already applied another architecture design. The Façade in this case study able to reduces transaction processing error rate appreciably.

As presented findings result demonstrate significance of design decisions and their impact for enterprise applications. The results can be utilized by application architects and designers to anticipate the behavior of an application depending on chosen design solutions. An initial suggestion for designers indicates that for the local deployment scenario, which is unlikely in a production environment, there are significant differences between legacy and refactored used Façade pattern. At the same time, in the case of managed controller and service layers, the differences between these system point Façade as a better solution, from the dependency and scalability management perspective.

5.2 Remarks and Future Work

The presented results are a good starting point for further pattern refactoring implementation. The results show differences between the legacy and the refactored system using Façade design pattern. However, the conducted tests were limited to only one design patterns and a specific technology, Java EE. Therefore, extended tests should be conducted and cover differences multiple patterns and technologies different from the Java EE technology, for example .NET technology and different several cases study. In addition, the tests should include a wider range of compared design patterns as architecture and other types patterns. The test scenarios also should cover all typical behavior of case study including all use cases of the application. The final objective would be a creation of a set of recommendations containing specific design patterns used on different layers of application and implemented in various technologies and variants.

(This page intentionally left blank)

REFERENCES

Aguiar, A., & David, G. (2009). *Transactions on Pattern Languages of Programming I. Lecture Notes in Computer Science* (Vol. 5770). http://doi.org/10.1007/978-3-642-10832-7

Ali, M., & Elish, M. O. (2013). A Comparative Literature Survey of Design Patterns Impact on Software Quality. *Information Science and Applications (ICISA), 2013 International Conference on*, 1–7. http://doi.org/10.1109/ICISA.2013.6579460

Booch, G. (2004). *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing.

Corporation, O. (n.d.). Java VisualVM Tool. Retrieved February 23, 2017, from https://visualvm.github.io

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2002). *Object Oriented Reengineering Patterns*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Eeles, P., & Cripps, P. (2010). *The Process of Software Architecting*. Addison-Wesley Professional.

Foundation, A. S. (n.d.). Apache JMeter. Retrieved February 23, 2017, from http://jmeter.apache.org

Fowler, M. (1997). *Analysis Patterns: Reusable Objects Models*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Ganesh, S. G., Sharma, T., & Suryanarayana, G. (2013). Towards a principle-based classification of structural design smells. *Journal of Object Technology*, *12*(2), 1–29. http://doi.org/10.5381/jot.2013.12.2.a1

Grady, R. B. (1992). *Practical Software Metrics for Project Management and Process Improvement*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Henderson-Sellers, B. (1996). *Object-oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Hunt, J. (2003). *Guide to the unified process featuring UML, Java, and design patterns*. London; New York: Springer.

IEEE Computer Society. (2009). IEEE Standard for a Software Quality Metrics

Methodology - IEEE Std 1061TM-1998 (R2009), *1998*.

IEEE Standard Glossary of Software Engineering Terminology. (1990). *IEEE Std 610.12-1990*. http://doi.org/10.1109/IEEESTD.1990.101064

ISO/IEC. (2001). *ISO/IEC 9126. Software engineering -- Product quality*. ISO/IEC.

ISO/IEC. (2011a). *ISO-IEC 25010: 2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Geneva: ISO.

ISO/IEC. (2011b). ISO/IEC 25023 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Measurement of system and software product quality. ISO/IEC. Retrieved from citeulike-article-id:10951573

ISO/IEC/IEEE Standard for Systems and Software Engineering - Software Life Cycle Processes. (2008). *IEEE Std 12207-2008*. http://doi.org/10.1109/IEEESTD.2008.4475826

Kerievsky, J. (2004). *Refactoring to Patterns*. Pearson Higher Education.

Khomh, F., & Gueheneuce, Y.-G. (2008). An Empirical Study of Design Patterns and Software Quality. *2008 12th European Conference on Software Maintenance and Reengineering*, 274–278. http://doi.org/10.1109/CSMR.2008.4493325

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Martin Lippert, S. R. (2006). *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley*. Retrieved from http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470858923.html

McConnell, S. (2004). *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press.

Meier, J., Farre, C., Bansode, P., Barber, S., & Rea, D. (2007). *Performance Testing Guidance for Web Applications: Patterns & Practices*. Redmond, WA, USA: Microsoft Press.

Opdyke, W. F. (1992). *Refactoring Object-oriented Frameworks*. University of Illinois at Urbana-Champaign, Champaign, IL, USA.

Pressman, R. (2010). *Software Engineering: A Practitioner's Approach* (7th ed.). New York, NY, USA: McGraw-Hill, Inc.

Priestley, M. (2003). *Practical Object-oriented Design with UML*. McGraw Hill Higher Education.

Rochimah, S., Akbar, R. J., & AVEROUSI, A. T. (2015). *Rancang Bangun Perangkat Lunak Sistem Informasi Akademik Generik Pada Modul Kurikulum*. JURNAL TEKNIK ITS, Institut Teknologi Sepuluh Nopember, Surabay.

Rochimah, S., Anggraini, R. N. E., & RAHMAN, H. (2015). *Rancang Bangun Sistem Informasi Akademik Generik Pada Modul Penilaian Menggunakan Pola Perancangan Hierarchical Model-View-Controller*. JURNAL TEKNIK ITS, Institut Teknologi Sepuluh Nopember, Surabay.

Rudzki, J. (2005). How Design Patterns Affect Application Performance -- a Case of a Multi-tier J2EE Application. In *Proceedings of the 4th International Conference on Scientific Engineering of Distributed Java Applications* (pp. 12–23). Berlin, Heidelberg: Springer-Verlag. http://doi.org/10.1007/978-3-540-31869-9_2

Sauer, F. (n.d.). Metrics Eclipse Plug-in. Retrieved February 23, 2017, from http://metrics.sourceforge.net/.

Society, I. C., Bourque, P., & Fairley, R. E. (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0* (3rd ed.). Los Alamitos, CA, USA: IEEE Computer Society Press.

Sommerville, I. (2010). *Software Engineering* (9th ed.). USA: Addison-Wesley Publishing Company.

Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). *Refactoring for Software Design Smells: Managing Technical Debt* (1st ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Suryn, W. (2014). Software quality engineering : a practitioner's approach. Retrieved from http://site.ebrary.com/id/10826717

Weisfeld, M. (2013). *The Object-Oriented Thought Process* (4th ed.). Addison-Wesley Professional.

Yuhana, U. L., Akbar, R. J., Agung, S., & Wijaya. (2016). *Rancang Bangun Kerangka Kerja Sistem Informasi Akademik Modular Berbasis Web Dengan Pola Arsitektur Hierarchical Model-View-Controller*. JURNAL TEKNIK ITS, Institut Teknologi Sepuluh Nopember, Surabay.

Yuhana, U. L., Akbar, R. J., & Nurwantoro, T. (2015). *Kerangka Kerja Sinkronisasi Basis Data Relasional Berbasis Web Pada Studi Kasus Sistem Informasi Akademik*. JURNAL TEKNIK ITS, Institut Teknologi Sepuluh Nopember, Surabay.

Yuhana, U. L., Anggraini, R. N. E., & Alfirdaus, B. A. (2015). *Rancang Bangun Perangkat Lunak Sistem Informasi Akademik Berbasis Web dengan Rancangan Modularitas dan Evolusi pada Modul Ekivalensi*. JURNAL TEKNIK ITS, Institut Teknologi Sepuluh Nopember, Surabay.

Yuhana, U. L., SUMINTO, G. P. N., & Anggraini, R. N. E. (2015). *Rancang Bangun Commercial Off The Shelf (Cots) Sistem Informasi Akademik Berbasis Web Pada Modul Kelola Pembelajaran*. JURNAL TEKNIK ITS, Institut Teknologi Sepuluh Nopember, Surabay.

(This page intentionally left blank)

AUTHOR BIOGRAPHY

Kholed Langsari was born in 1988, near sub-district Budi, district Yala. He grew up in Yala district, Thailand, eventually studying at The Phatna Witya School, a locally acclaimed, private high school in 2006.

He attended Yala Islamic University from 2007 to 2011, and graduated in Bachelor of Technology Program in Computer Science (International Program). His was design and implement the online register information system for Yala Community College, at Yala province. He was also one of the first hackers on campus who success wrote the code to broke down university internet access time countdown.

He was support in scholarship by DIKTI, Indonesia and Fatoni University, Thailand for master degree education from 2014 to 2016. He has given numerous invited talks and tutorials.

Kholed has five years of experience in education and is currently teaching Computer Science at the Fatoni University in Pattani province, Thailand where he has been a faculty member since 2011. He is the instructor on undergraduate topics in computer networking and computer science.

His research interests span both software engineering and computer networking. He has collaborated actively with researchers in several other disciplines of computer science, particularly computer networking on problems at the computer server, and enterprise networking system.

Feel free to contact to him via email address: langsaree@gmail.com, phone number: (+62)838-1151-3404 or (+66)83-192-0711 and personal blog: http://ikholed.wordpress.com.