International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

Refactoring Legacy Software for Layer Separation

Alireza Khalilipour
Sama Technical and Vocational Training College, Islamic Azad University, Mahshahr Branch, Mahshahr, Iran.
a.khalilipour@mhriau.ac.ir

Moharram Challenger*
Department of Computer Science, University of Antwerp, Middelheimlaan 1, 2020 Antwerp; and Flanders Make, xzw, Belgium.
moharram.challenger@uantwerpen.be

Mehmet Onat
R&D Center, Ford Otosan Inc., Istanbul, Turkey.
monat2@ford.com.tr

Hale Gezgen
R&D Center, KocSistem Information and Communication Services Inc., Istanbul, Turkey.
hale.gezgen@kocsistem.com.tr

Geylani Kardas
International Computer Institute, Ege University, Bornova 35100, Izmir, Turkey.
geylani.kardas@ege.edu.tr

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Abstract One of the main aims in the layered software architecture is to divide the code into different layers so that each layer contains related modules and serves its upper layers. Although layered software architecture is matured now; many legacy information systems do not benefit from the advantages of this architecture and their code for the process/business and data access are mostly in a single layer. In many legacy systems, due to the integration of the code in one layer, changes to the software and its maintenance are mostly difficult. In addition, the big size of a single layer causes the load concentration and turns the server into a bottleneck where all requests must be executed on it. In order to eliminate these deficiencies, this paper presents a refactoring mechanism for the automatic separation of the business and data access layers by detecting the data access code based on a series of patterns in the input code and transferring it to a new layer. For this purpose, we introduce a code scanner which detects the target points based on these patterns and hence automatically makes the changes required for the

*Corresponding author

layered architecture. According to the experimental evaluation results, the performance of the system is increased for the layer separated software using the proposed approach. Furthermore, it is examined that the application of the proposed approach provides additional benefits considering the qualitative criteria such as loosely coupling and tightly coherency.

Keywords: Layered Software Architecture; Code Refactoring; Software Layers Separation; Software Modernization; Data Access Layer; Business Layer.

1. Introduction

Layering as a structural approach for increasing the abstraction level of a system is used not only in computer engineering, but also in many other disciplines. In software systems, this method is used to dominate the complexity of systems, such as enterprise systems [1], composite content applications [2] and cloud-based systems [3].

Layered software architectures enable developers to group their code in different layers. In this way, various tasks will be categorized in different layers which can lead to increase the development performance, decrease the cost of maintenance and development and simplify the distribution of the software over nodes of a network e.g. to create a distributed system [1]. Also, separation of the layers makes it possible to move the layers as services to a cloud which constitute a Software as a Service (SaaS) [4] for performance improvement.

The layered software architecture mainly targets business information systems which include a database to save users' data permanently and execute some processes in storing/retrieving these data into/from the database. Generally, an information system has three layers [5]: Presentation, Logic (Process or Business), and Data Access. The layered architecture suggests that different parts of the software code are placed into these layers which brings some advantages. First, the dependency of different parts of the code will decrease which results in simplicity of making changes in each layer. Second, with distributing each layer on a node of a network, it is possible to distribute and balance the server load, especially in web servers.

Despite all the advantages discussed for this architecture, in the legacy systems, many developers have used integrated architectures such as Windows Distributed interNet Applications (DNA). In this architecture, the codes for process/logic layer and Data Access Layer (DAL) are located in a single layer called Business Layer (BL). Nowadays, there is good number of running applications based on this kind of technologies. However, these systems have a high risk of crashing, due to both the centralization of computation load on the server hosting this layer and the lack of supporting rapid changes on the BL. Therefore, in legacy systems, most of the processing and business logic of the software is in a monolithic form. The monolithic systems face risks and problems over time which lead the motivation for (automatic) refactoring of these systems. Refactoring is a well-known software maintenance activity that improves the software quality by changing the structure of the code or architecture in a variety of ways.

The purpose of the refactoring is to achieve the quality features such as reusability, usability, and benefits such as understandability, ease of finding software bugs, and obtaining a faster development process [6]. Besides, software needs to change over time due to new requirements. But the complexity of legacy and monolithic systems makes these change difficult [7]. Therefore, refactoring will inevitably be the solution in front of us. Legacy systems continue to play an important role in the implementation of information systems in organisations.
Many large organisations still rely heavily on these systems in delivering critical services. Legacy systems are important assets of organisations as they contain important business logic and data over several years. Although these systems are critical to the business yet organisations have to face technical difficulties and unnecessary expenses in maintaining the systems. In order to continue providing quality services in line with the global changes, legacy systems need to be refreshed through modernization [8]. Moreover, other disadvantages of monolithic architecture such as being non-scalable and non-reliable also force us to refactoring [9]. Given the difficulty of change and the difficulty of debugging monolithic systems, this can also be a motivation to refactor them [10]. Another motivation to refactor monolithic systems is the use of new technologies such as the microservices. The microservice architecture is widely used to overcome the restrictions of monolithic legacy systems, such as limited maintainability and reusability. Migration to a microservice architecture is increasingly becoming one of the foci of software engineering research [11].

As also stated in [12], many new system architectures are constructed based on the previous applications which are either re-engineered or wrapped on the legacy systems. Hence, in most situations, there is no need to develop a new system (from scratch) and refactoring can be a more appropriate option. Refactoring can be done manually or automatically. However, manual refactoring is inefficient, tends to be error-prone, and mostly challenging as expected. Even small changes (only a few lines of code) have a significant chance of introducing a bug. So, there is a need for the automatic refactoring [13].

To this end, in this paper, a code scanner is proposed which can receive the input code, detect and separate the software layers in an available information system. Our focus is the separation of the BL and the DAL, since usually these layers constitute a large part of the code in an information system and improving them can have a high impact on the performance of the overall system.

The remaining of the paper is organized as follows: In section 2, the background required for the paper is presented. Section 3 gives the related work. The architecture and the operational phases of the scanner are discussed in section 4. Section 5 evaluates the proposed methodology. Finally, the paper is concluded, and some future studies are suggested in section 6.

2. Background

In a layered system, including layered software architecture, each layer provides service(s) for its upper level and receives service(s) from its lower layer. To reduce the connections between the layers, services which are strongly related to each other (tightly coherent) should be located in the same layer. However, the provided layers should also be loosely coupled and have a weak connection to reduce the system communication load. A layered system is based on a hierarchy and in this manner the layering can be considered as a logical grouping for the subsystems based on their functionality. To this end, a business information system can have the following hierarchy [14]:

- Presentation/Interaction Layer: Tasks and functions related to the user as the highest level. All display components are placed in this layer to display the extracted information to the user, such as ComboBox, TextBox, ListBox, etc.
- Business/Logic Layer (BL): Tasks and functions related to the problem domain as the middle level. The modules that make up the main logic of the software fall into this layer, such as payroll calculations, audits, comparisons, sorts, search algorithms, schedules, work order priority, and so on.
- Data Access Layer (DAL): Tasks and functions for data transportation. The commands and modules that communicate between the business layer and the final data are located in this layer, such as the types of data access components (local or remote) that are responsible for sending and receiving data to/from the data source.
- Data Layer: The data and the data management system as the lowest level. In this layer, there are data source and their management, such as types of databases, tables, views, triggers, etc. Low-level operations are performed on these components (such as executing SQL commands) in this layer.

In Information Systems (IS), a large part of the code is related to the interaction between the software with a repository, since each IS has at least one data resource, e.g. a repository, to use the data. This code which is called the data access code has a great impact on system performance. For this reason, the providers of the components related to this part of the software work on improving the capability of this part, since it can naturally affect the overall system performance [15]. Over the last two decades, the developers used ADO.NET and JDBC as tools for data access since the introduction of .NET and J2EE as the promising platforms for enterprise systems [16]. Although, these components were among the complete tools for data access, the improper use of them has become the reason for some problems in the functionality of IS. Websites with many users and transactions, e.g. application servers, educational portals and banks, are examples of such systems. Their problems with low performance or even systems crashing are not because of malfunctioning of those components, but improper use of them in these systems [17].

The DAL fills the gap between the business/process layer and the data layer in the layered architecture. The BL includes the code for process and logic of the system which is required for system data that is in databases or other information resources. So, to retrieve the data from these resources, the software should have some code for data access. Also, after the process, the software needs to store the information into the resources which requires some data access codes.

Having fewer layers in the architecture, more code will be integrated in the unified layers, which increases the probability of the system failure in those layers. The problem of separating the layers, especially the DAL, is also important from the security point of view. To this end, it is possible to apply security mechanisms in the code level over these layers.

Regardless of the scale and the complexity of the software system and utilized middleware, data access is realized by the following steps [18]: Connection to the data resource, preparing the command (e.g. SQL command), executing the command and receiving the result. These steps are illustrated in Figure 1 in the form of a sequence diagram. This diagram describes the protocol for data access in the traditional layering model for software systems.

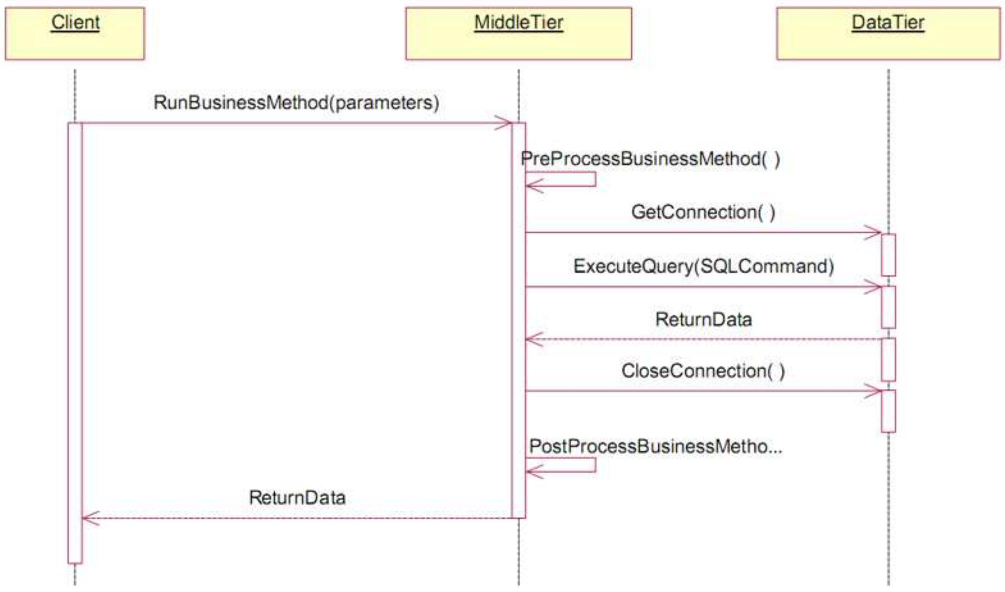


Figure 1. The sequence diagram describing the steps of data access in the traditional layering model for software systems

The above-mentioned steps depicted in Figure 1 are elaborated as follows:

- Connection to the data resource: To connect to a data resource, a connection object is usually used. This object includes the required information for making the connection, such as the address/URL of the server, server name, port number, username, and password. Using this object, the requests are delivered to the data resource and the replies are received. To realize data access, a connection should be opened and after finishing the access, it should be closed. The connection generally includes these steps: Setting the connection parameters, opening the connection, and closing the connection.

- Command preparation: After establishing the connection, the desired query should be sent to the data resource. But before submitting the query, it should be encapsulated in an object. To this end, a command object is used. This object executes the query in the data resource and returns the result using the established connection. The preparation of the command object consists of selecting the connection object and specifying the SQL command.
- Command execution and returning the result: After preparing the command object, the command should be executed using this object. The object usually has different methods, due to a variety of SQL commands. Generally, these methods have the ability of executing the following commands: Select command, Update commands (including delete, update, insert), Data Definition Language (DDL) commands (such as create table and so on).

Finally, after execution of the command, the command object returns the result (if there is any) in the form of a data structure, such as record set, which can be used in the programming languages.

As also indicated in the introduction, the legacy systems based on a monolithic architecture encounter many problems such as no-scalability, unreliability, high maintenance costs, difficult debugging, and hence refactoring to the separation of their above described layers provides a convenient way to overcome their problems.

3. Related Work

To do code refactoring and apply compiler techniques both for the increase in software performance or the separation of the code, there are many studies which can be categorized under the names of code parallelizing and clustering, and migrating from legacy to cloud and SOA.

3.1. *Parallelizing the Legacy Software and Software Clustering*

There are studies in which the authors try to refactor or apply layering for parallelizing or distributing software. However, these studies have their criteria for the layering, and they do not consider the principles of layered software architecture. For example, some of the studies try to generate a distributed code from the serial code, such as Parsa and Bushehrian's study in 2008 [19]. Dolz et al. [20] presents a framework for discovering pipeline and farm parallel patterns of a sequential system and converting them to a Generic Parallel Pattern Interface (GrPPI). Alsubhi et al. [21] present an architecture for converting legacy codes into parallel web service codes based on MPI, CUDA, OpenMP and OpenCL models. In the proposed method, two goals are pursued, one is the conversion of sequential to parallel codes and the other is to provide parallel codes as web services.

Parsa and Bushehrian [22] cluster the object-oriented programs in two phases to speed up with increasing their concurrency level. In phase one, the program instructions are re-ordered in a way that the distance between a calling instruction and its dependent instructions become maximum. In phase two, a clustering algorithm is used to cluster the modules provided in the previous phase with the aim of distributing and executing the clusters with the maximum level of concurrency. Similarly, Muhammad et al. [23] propose an approach for automatic modularization of the clusters which are resulted from object-oriented programs. With this clustering algorithm, the related objects in the program are grouped in a cluster which is a challenge when different criteria are considered. Muhammad et al. [23] evaluate these criteria and suggest some proposals for various situations. However, the proposed methodologies in [22] and [23] are mostly for an object-oriented code and try to cluster the objects of the software to run the initial serial code in a parallel manner. These approaches are not suitable for layered systems, since they only address the relation between objects, and they do not consider software layers in their clustering.

Alkhalid et al. [24] propose an approach to distribute the load in the package level. To this end, a Computer Aided Support is provided to inform the programmer during program design about the level of internal and external connection of a package with other packages. The problem with this approach is that the separation of packages (or layers) is provided with the direct intervention of the programmer. On the other hand, Millham [25] proposes a method to convert the legacy single layer system into a two-layer system. In fact, most of the above-mentioned noteworthy studies aim at converting one-layer systems into two-layer systems while the aim of our approach is to separate BL and DAL.

The study presented by Andreopoulos et al. in [26] introduces an algorithm for layer clustering, but the layers targeted in this algorithm are not aligned with the criteria of the layered software architecture; and utilizing this kind of algorithm in the development of layered information systems will not guarantee the goal of layered software architecture. The reason is that they try to cluster to optimize the performance and do not consider the architectural layers in the clustering.

3.2. *Software Modernization and Refactoring*

Code refactoring applies some changes on the source code by preserving the behaviour of software [27] which may also lead to re-structuring software. For example, Santos et al. [28] consider the refactoring as an appropriate solution for the modernization of legacy software systems. By conducting a case study, the effect of refactoring on the improvement of the code maintainability has been studied by Wahler et al. [29].

Moreover, Zimmermann [30] examines the issue of refactoring at the architectural level and introduces an architectural refactoring as a new strategy for improving maintainability. Similarly, in [31], Kandukuri modernizes a legacy software by re-engineering the software using model extraction and transformation.

In addition, there are studies which address the increase of software performance and quality with applying some changes in the environment or in the codes. In fact, the main goal of these studies is to re-use the software components. For instance, there are some studies in which some experiments are conducted to change the type of services or service calls with the aim of performance increase or reuse. Various approaches (e.g. [32], [33], [34], and [35]) are suggested for changing the synchronous calls to asynchronous calls. Although these techniques have some improvements in system functionality, they do not make any change in the architecture or layers of the system. Also, in [36], Balis et al. offer an approach to convert the legacy services into web services. The aim of this approach is only to increase the reuse in the software, and it does not have any suggestions for the separation of layers.

3.3. *Legacy to SOA*

Some methodologies and techniques (such as the studies in [37], [38], [39], and [40]) are provided to convert the architecture of legacy systems into SOA [41]. However, unlike our approach, these approaches do not focus on layering the software, instead, they only focus on converting into SOA.

Also, there are few studies in which, as in our study, the authors aim to provide software layering. For example, Pereira et al. [42] propose a framework for implementing a software BL in a way that it is also re-usable. Although one of the indirect goals of this study is the separation of layers, the proposed method is not in the compiler level, but in the implementation level. This type of approaches is useful for new systems with layered architecture but is not useful for improving legacy systems with classical architectures as being targeted in our study.

Finally, Heckel et al. [43] introduce an approach for converting legacy systems to layer-based and SOA-based systems. In this study, the authors introduce an approach to convert traditional one or two-layered systems to three layered ones. Their approach includes 4 steps: Annotation, Reengineering, Redesign, and Forward Engineering. The code is annotated based on pre-defined categories such as input instructions, business instructions, and data access instructions. Then, in the Reengineering phase, a graph is extracted from the code. This graph is Redesigned considering the node dependency; and finally, the new code is generated in the Forward Engineering phase. However, when we compare our study with the study of Heckel et al. [43], we can see that the annotating the code is performed in a manual way by Heckel et al. [43] which is a time-consuming task. So, Heckel et al.'s approach is a semi-automatic while our approach is fully automatic. Finally, in [43], the aim is not distributing the layers and the approach does not support this goal while code (layer) distribution is one of the main goals in our approach.

4. The Architecture of the Code Scanner

The scanner proposed in our study does not only parse the code, but it also converts it to target layers. In other words, the scanner, as can be seen in Figure 2, receives the middle layer of an information system including data access and business codes as input and then scans them to detect the pre-defined patterns. Then, it separates the code and as output it gives the middle layer in the form of two separated layers, data access and business.

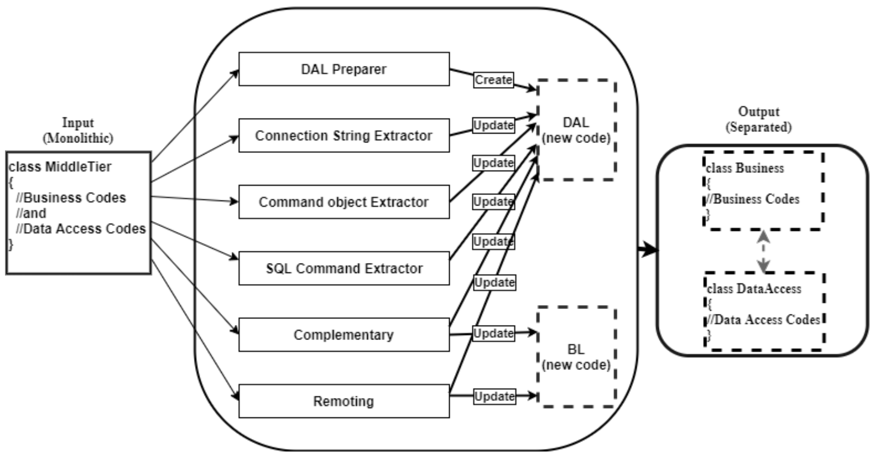


Figure 2. The proposed architecture for the scanner

As it can be seen in the scanner architecture, the scanner identifies the necessary conversion code in 6 phases and converts the initial code into separate layers. These phases are *DAL Preparer*, *Connection String Extractor*, *Command Object Extractor*, *SQL Command Extractor*, *Complementary* and *Remoting*. These phases are discussed in detail in this section.
This scanner is not limited to any language with specific grammar and fixed rules, instead, it is designed to be customizable. Therefore, any legacy system can be the input of this scanner, provided that the rules related to data access code in the system are given as input to the scanner.

Basically, the operation of accessing data is moved to another layer. The sequence diagram showing data access, based on the newly generated code, is given in Figure 3. Comparing the data access request in the traditional approach (Figure 1), and the new approach (Figure 3), in Figure 1 the request is directly sent to data resource, but in Figure 3, with layer separation, the request is sent from the BL to the DAL and it is delivered to the data resource indirectly.

To reach this goal, the code scanner separates the code which fulfills the operation of data access based on the proposed approach and moves them to another layer. However, the remaining code needs to be manipulated to keep the connection

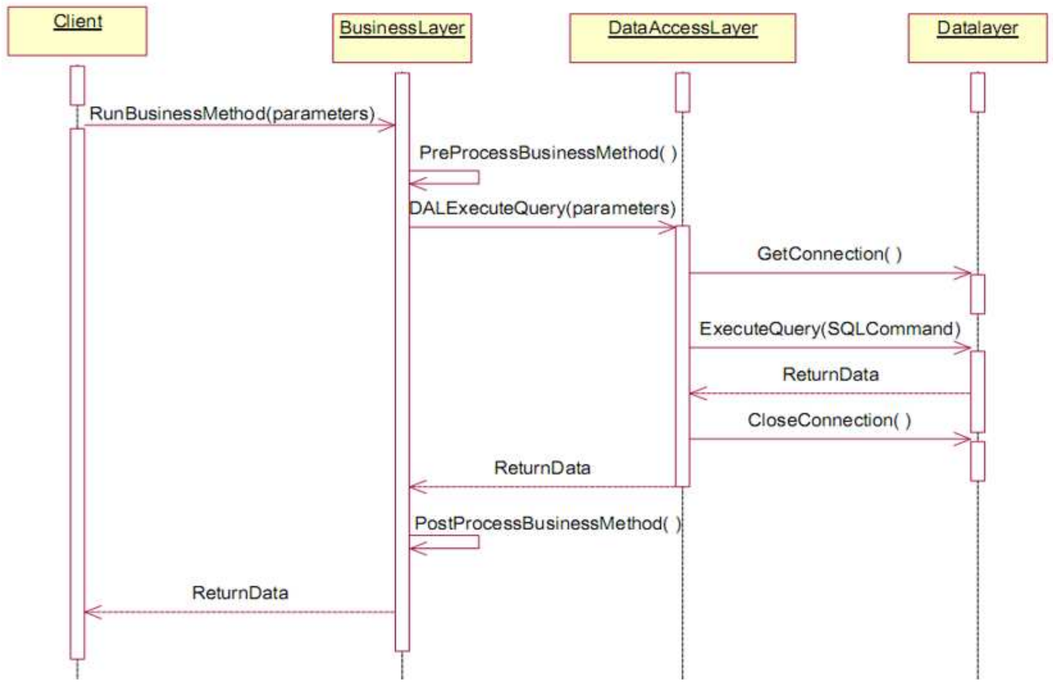


Figure 3. The sequence diagram for data access in the proposed layering model

between these two new layers correctly. The critical issue is that the scanner should not change the logic of the source code which is considered in the design of the approach. According to the operations which should be performed by DAL, we can divide the phases of the approach for scanning and separation as follows:

Phase 1: Preparing the DAL. This phase includes the initial steps which the scanner should do for separating the code. The main task of this phase is creating the new DAL including related package and name space.

Phase 2: Scanning and code conversion for connection operations. In this phase, the scanner detects the codes related to connecting operations and makes required changes on them.

Phase 3: Scanning and code conversion for preparing command object. This phase provides the operations which are needed for command object.

Phase 4: Scanning and code conversion for executing the commands via command object such as SQL-SELECT, SQL-UPDATE, and SQL-DDL. The instruction of the program which includes one of the SQL commands should be executed via command object on database. The operations of moving these commands into DAL are realized in this phase.

Phase 5: The complementary operations. After moving the required commands from the BL into the new layer, the DAL, the complementary operations should be done on both layers. For example, removing the remaining commands in the BL and adding new commands into the DAL such as closing the connections. These tasks are fulfilled in phase 5.

Phase 6: Adding remoting capability to layers. Finally, the scanner should add remoting capability to layers to provide the ability of their connection in a network. This phase covers the scanning operations to realize this goal.

In the following, the implementation details of these phases in addition to the approach for automatically generating the separated code are discussed.

A template of a legacy system is given as an input to the scanner. Although, this template can vary slightly in different cases, the main elements are the same and with slight adjustments in the scanner, the functionality of layer separation can be realized successfully. The input code in our case is in Java language representing mid layer of an information system. This layer consists of a combination of process and data access codes.

The scanner detects different elements of the program statement after it extracts the parse tree of the statement. Then, using the proposed approach, it transforms the statement into different statements in separated layers.

In our case, the component which is used for accessing the data is the Java Database Connectivity (JDBC); and the Remote Method Invocation (RMI) mechanism is used for establishing connection between the layers. The execution of the scanner is presented in the Tables 1 to 6 for each phase of the following approach.

Phase 1: Creating a data access class

The scanner, in this phase, constitutes the layers and creates a class named DALClass, see Table 1. Since this class will be located in a computation node and perform the required computations, it will not be serializable. Serializable objects are transmitted completely to the client and this feature does not help in load balancing.

This phase is elaborated as follows:

- Create a package named DAL. This is the structure for DAL.
- Create a package named BL
- Create a class named BLClass in BL package and move the initial code there. This is the structure for BL.
- Create a non-serializable class named DALClass in DAL.

Phase 2: Preparing the connection object

In this phase, the parse tree of Connection statement is extracted and then the

Table 1. Input and output of the scanner for phase 1.

before	`class MiddleTier {` `//Business` `//and` `//Data Access codes` `}`	
after	`package BL;` `class BLClass {` `//Business` `//and` `//Data Access codes` `}`	`package DAL;` `class DALClass {` `//Empty` `}`

layers are constructed using the required elements of the tree, circled with the red color in Figure 4.

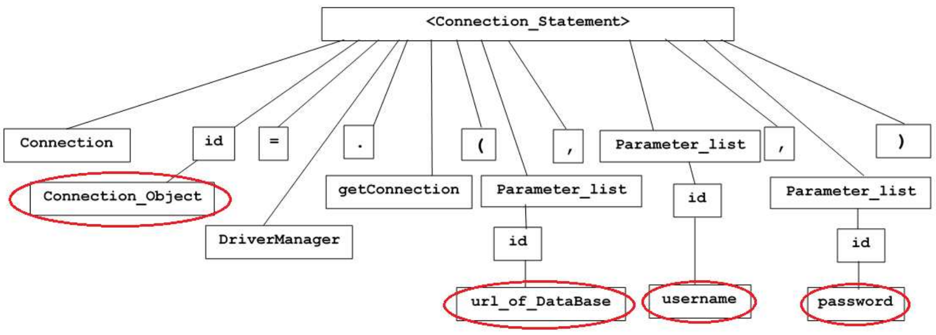


Figure 4. The parse tree of the Connection statement

Also, the objects created from a serializable class are sent and received by the value in the network. The results of these operations are demonstrated in Table 2.

This phase is elaborated as follows:

- Create a serializable class named CONNECTION with the following pattern and add it to BL and DAL packages.

```
class CONNECTION extends Serializable {
    String DriverName;
    String URL;
    String UserName;
    String Password;
}
```

- All the connection objects in BL are converted to an object of the CONNECTION class.

Table 2. Input and output of the scanner for phase 2.

before

```
class MiddleTier {
 void TestMethod(){
 String driverName = "oracle.jdbc.driver.OracleDriver"; // for Oracle
 Class.forName(driverName);

 String serverName = "linux.ce.xyz.edu";
 String portNumber = "1234";
 String dbName = "dbtest";
 String username = "username";
 String password = "password";
 String url = "jdbc:oracle:thin:@" + serverName + ":" +
    portNumber + ":" + dbName; // for Oracle

 Connection connection=DriverManager.getConnection(url,username,password);
 }
}
```

after

```
package BL;

class CONNECTION extends Serializable {
   string  DriverName;
   string  URL;
   string  UserName;
   string  Password;
  }

class BLClass {
 void TestMethod( ){
  CONNECTION connection = new CONNECTION();
  String driverName =
  "oracle.jdbc.driver.OracleDriver";//for Oracle
  //Class.forName(driverName);

  String serverName = "linux.ce.mahiau.edu";
  String portNumber = "1234";
  String dbName = "dbtest";
  String username = "user name";
  String password = "password";

  connection.DriverName= driverName;
  connection.URL= url;
  connection.UserName= username;
  connection.Password= password;
  //Connection connection =
DriverManager.getConnection(url,
  //username, password);

  }
}
```

```
package DAL;

class CONNECTION
extends Serializable {
   string  DriverName;
   string  URL;
   string  UserName;
   string  Password;
  }

class DALClass {
//Empty
}
```

- All the connecting parameters in the source code are encapsulated with a proper format of the new CONNECTION class.
- All the instructions leading to a connection are removed from BL.

Phase 3: Preparing a command object

The aim of this phase is to execute the instruction in the new layer, DAL. So, the command object should be moved to this layer. To this end, the command object is re-organized, and its main parameters are marshaled in a string and sent to the new layer to be executed.

In this phase, the parse trees of different Command statements are extracted and then the layers constructed in the previous phase are extended using the required elements (marked with the red color) of the trees, Figure 5. It is worth noting that the numbers beside the parse trees in this figure indicate the order of the statements in the 'before' section of the Listing 3. Figure 5 shows the parse tree for executeQuery, among the others. However, the scanner can create the parse tree for executeUpdate and executeDDL commands in a similar manner.

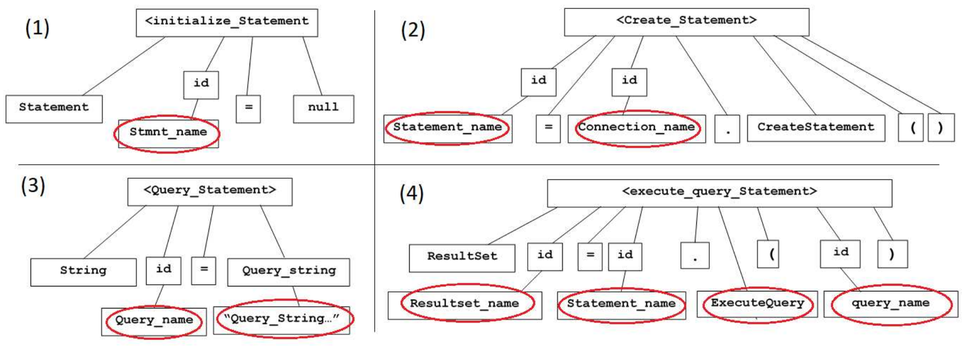


Figure 5. The parse trees of Command statements

The result is shown in Table 3. The details of this phase in the approach are described below:

- Create a new serializable class named COMMAND with the following pattern and add it to BL and DAL packages:

```
class COMMAND extends Serializable {
    CONNECTION CON; // Comes from the previous phase
    String Query;
}
```

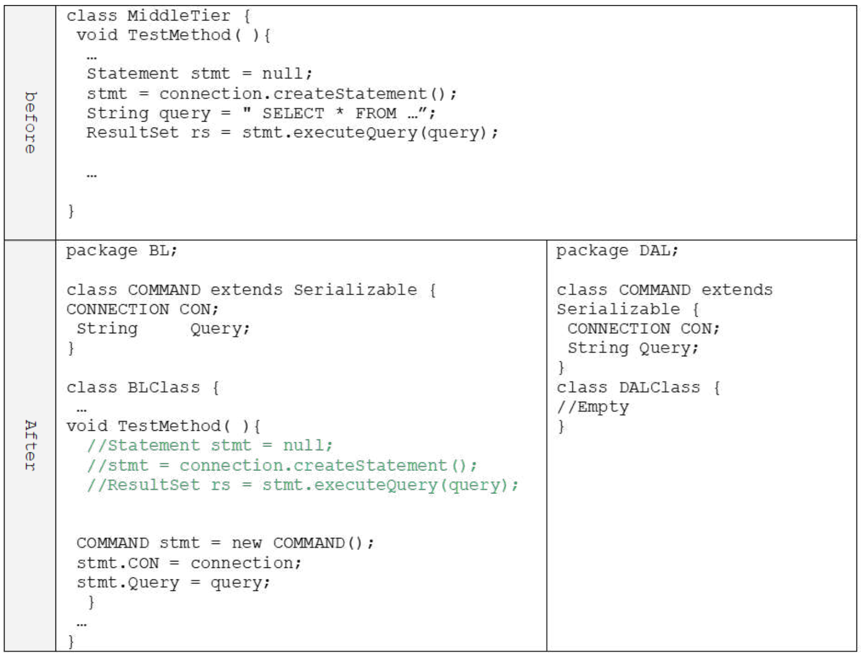
- Convert all the command objects in BL to objects of COMMAND class.
- Encapsulate all the parameters of command object which are in BL (i.e. connection object and the query) into COMMAND class

Phase 4: Executing a SQL command via COMMAND object

Since the execution of the commands should be done in the DAL layer, all the commands which are representing SQL commands in BL should be removed and requests for their execution should be sent to DAL. Therefore, DAL should have an interface layer which covers all the commands including SELECT, UPDATE, and DDL. The result of these modifications on the source code and the new code are illustrated in Table 4 for the example.

Finally, the elaborated tasks of phase 4 in the approach are stated below:

Table 3. Input and output of the scanner for phase 3.



before

```
class MiddleTier {
 void TestMethod( ){
  …
  Statement stmt = null;
  stmt = connection.createStatement();
  String query = " SELECT * FROM …";
  ResultSet rs = stmt.executeQuery(query);

  …

}
```

After

```
package BL;

class COMMAND extends Serializable {
CONNECTION CON;
 String     Query;
}

class BLClass {
 …
void TestMethod( ){
  //Statement stmt = null;
  //stmt = connection.createStatement();
  //ResultSet rs = stmt.executeQuery(query);


 COMMAND stmt = new COMMAND();
 stmt.CON = connection;
 stmt.Query = query;
  }
 …
}
```

```
package DAL;

class COMMAND extends
Serializable {
 CONNECTION CON;
 String Query;
}
class DALClass {
//Empty
}
```

- Create a reference of DAL class, dal, in BL: This reference is needed when the commands are called.
- Replace all of the SELECT commands in BL with calling SELECT() service using dal: dal.SELECT(COMMAND);
- If the result of executing SELECT command in BL is stored in a non-serializable object, an alternative serializable object should be replaced.
- All the INSERT, DELETE, and UPDATE commands should be replaced with UPDATE service using dal: dal.UPDATE(COMMAND);
- All the DDL commands such as CREATE TABLE, should be replaced with DDL service using dal: dal.DDL(COMMAND);
- Add a method named SELECT() with the following signature into DALClass: RecordSet SELECT(COMMAND)
- Add a method named UPDATE() with the following signature into DAL-Class: void UPDATE(COMMAND)
- Add a method named DDL() with the following signature into DALClass: void DDL(COMMAND)

The function of the SELECT, UPDATE, and DDL methods are:

- Extracting SQL command and connection string from COMMAND object

Table 4. Input and output of the scanner for phase 4.

before	class MiddleTier {
	 void TestMethod( ){
	  …
	  Statement stmt1 = null;
	  Stmt1 = connection1.createStatement();
	  String query1 = " SELECT * FROM …";
	  ResultSet rs = stmt1.executeQuery(query1);
	
	  Statement stmt2 = null;
	  Stmt2 = connection2.createStatement();
	  String query2 = " INSERT/UPDATE/DELETE …";
	  stmt2.executeUpdate(query2);
	  Statement stmt3 = null;
	  Stmt3 = connection3.createStatement();
	  String query3 = " SQL-DDL …";
	  stmt3.execute(query3);
	  …
	}	

after	package BL;
	import DAL.*;
	
	class BLClass {
	 …
	void TestMethod(){
	  IDALClass dal;//will complete in step 6
	  //ResultSet rs =
	  stmt1.executeQuery(query1);
	  ResultSet rs = dal.SELECT(stmt1);
	
	  //stmt2.executeUpdate(query2);
	  dal.UPDATE(stmt2);
	
	  //stmt3.execute(query3);
	  dal.DDL(stmt3);
	
	 }
	 …
	}
		package DAL;
		
		class DALClass }
		public CachedRowSet SELECT(COMMAND CMD)
		    }
		//Resolving CMD
		//Execute SELECT command
		//return Result
		    {
		public void UPDATE(COMMAND CMD)
		    }
		//Resolving CMD
		//Execute INSERT/DELETE/UPDATE command
		 }
		public void DDL(COMMAND CMD)
		 {
		//Resolving CMD
		//Execute DDL command
		 }
		}

- Executing SQL command
- Returning the result to BL in the form of a serializable object from RecordSet family (for SELECT method)

Phase 5: Complementary operations

This phase includes the following steps:

- Remove all the commands to close the connections and dispose the command objects in BL.
- Open the required connections at the beginning of DALClass methods
- Close all the connections and dispose all of the command objects at the end of DALClass methods.

The results of executing this phase of the scanner over an example are shown in Table 5.

Phase 6: Adding Remoting capability to the layers

Table 5. Input and output of the scanner for phase 5.

before

```
class MiddleTier {

  void TestMethod( ){
    …
    connection.Close();
    stmt.close();
    …
  }
}
```

After

```
package BL;
class BLClass {
 …
void TestMethod( ){
  …
 //connection.close();
 //stmt.close();
  …

 }
 …
}
```

```
package DAL;

class DALClass {
  public CachedRowSet SELECT(COMMAND CMD){
    //Resolving CMD
    connection = DriverManager.getConnection(url,
      username, password);//open connection

    //Execute SELECT command
    connection.close();
    stmt.close();
    //return Result
  }
  public void UPDATE(COMMAND CMD){
    //Resolving CMD
    connection = DriverManager.getConnection(url,
      username, password);//open connection

    //Execute INSERT/DELETE/UPDATE command
    connection.close();
    stmt.close();
  }
  public void DDL(COMMAND CMD){
    //Resolving CMD
    connection = DriverManager.getConnection(url,
      username, password);//open connection

    //Execute DDL command
    connection.close();
    stmt.close();
  }
}
```

Due to the separation of BL and DAL layers and the possibility of their parallel execution in the form of two different processes on two different computation nodes, their communication should be remote [16]. To this end, phase 6 fulfills some required steps as follows and the result of the scanner in this phase is depicted in Table 6 and 7.

This phase includes the following steps:

- Generating the interface from DALClass according to the Table 7.
- Generating the Client-Stub and Server-Skeleton from the interface
- Creating the server program to manage DALClass objects and register in the name registry
- Add Client-Stub to BL
- Add Server-Skeleton to DAL

It is worth noting that the Client_Stub and Sever_Skeleton are generated by

Table 6. Input and output of the scanner for Phase 6.

before

```
class MiddleTier {
 void TestMethod( ){
  …
 connection.Close();
 stmt.close();
  …
 }
}
```

After

```
package BL;
import java.rmi.*;

class BLClass {
 …
void TestMethod( ){
  …
// Install a security manager!

System.setSecurityManager(new
RMISecurityManager());
       try {
  // Get a reference to the
remote object.
  IDALClass dal =
(IDALClass)Naming.lookup("rmi:
//serverhost/DALClass");
System.out.println("Bound to:
" + server);
 …
catch(Exception e) {
         e.printStackTrace();

 }
 …
}
```

```
package DAL;
import java.rmi.*;

class DALClass {
public CachedRowSet SELECT(COMMAND CMD)
 {
  …
 }
public void UPDATE(COMMAND CMD)
 {
  …
 }
public void DDL(COMMAND CMD)
 {
  …
 }
public static void main(String[] args) {
  // Install a security manager!
  System.setSecurityManager(new
RMISecurityManager());
  try {
    // Create the remote object.
   DALClass obj = new DALClass("DALClass");
    // Register the remote object as"DALClass".
Naming.rebind("rmi://serverhost/DALClass", obj);
   System.out.println("DALClass bound in
registry!");
       }
       catch(Exception e) {
   System.out.println("DALClass error: " +
                           e.getMessage());
   e.printStackTrace();
 }
}
```

Table 7. The Interface for accessing the data

```
//DALClass Interface
import java.rmi.*;
/* DALClass Interface. */
public interface IDALClass extends Remote {
    public CachedRowSet SELECT(COMMAND CMD) throws RemoteException;
    public void UPDATE(COMMAND CMD) throws RemoteException;
    public void DDL(COMMAND CMD) throws RemoteException;
}
```

the compiler and added into BL and DAL. The idea of stub and skeleton is adopted from RPC which is used in many remoting mechanisms. Generally, stub and skeleton are generated from the interface of the considered class and added to the code of the client and server [16].

5. Experiment Process

5.1. *Definition*

In the following experiment, which is structured according to the principles of conducting experiments in software engineering [44], we seek to answer the following research question: Does the refactored code run more efficient than the legacy version?
The goal is to show the efficiency of the proposed method. In this regard, our objective is to evaluate the proposed method for software engineering schematic metrics as well as to perform an experiment on a case study and review the results before and after the change (layer separation).

5.2. *Planning (Design of Experiment)*

In order to test and answer the previous question, we first evaluate the proposed method in terms of objectivist criteria. Evaluation of these criteria as well as the outputs of the experiment indicates the correctness of the proposed method. Also, in order to achieve the next goal, we define independent and dependent research variables and by conducting an experiment on an information system in two modes (monolithic and multilayer), we will see whether statistically better performance is obtained from the proposed method at runtime. In this experiment, we first measure the response time of the single-layer system (scanner input) before making changes with a different number of requests. Then, after making changes by the scanner on the original code, we perform the same test on the modified code (multilayer). Details of the test conditions are given in Section 5.4. Also, the observations show that some of the object-oriented metrics are also met on the new code, which we are elaborated in the next section.

5.3. *Object Oriented Metrics Investigated*

A significant number of object-oriented metrics for evaluating object-oriented systems have been introduced in the literature [45], and we benefit from some of the most widely used ones in also evaluating our proposed method.
McCabe Cyclomatic Complexity (CC):
This metric indicates the maintainability of a code by measuring its complexity. To measure complexity, in this study, the relation “connections - nodes + 2” is used, in which nodes represent the instructions and connections represent the connection and sequence between the instructions. The larger value of this relationship, the more complex it is, resulting in lower maintainability. In the proposed method, due to the separation of layers from each other, many commands and connections are separated from each other and placed in separate modules, which makes the complexity of

the resulting code (multi-layer) less than the original code (single layer) and its maintenance improves.

Source Lines of Code (SLOC):

This metric indicates the number of physical lines of the program without blanks and comments. In our proposed method, since the functionality of the software is not affected before and after the change, as a result, we will not see a significant difference in the number of lines after the change, because our method is more about moving codes than adding new codes. Of course, after extracting the new layers, a number of commands will be added to establish communication between the layers, which will be very small compared to the main lines of the program.

Weighted Method per Class (WMC):

This metric is a cumulative measure of the complexity of methods within the classroom. If we examine the complexities of methods by metric CC, then the complexity of the class is: Sum(CC). In the proposed method, the complexity of the classes after conversion is less than before because the complexity of the methods has decreased.

Coupling between Objects (CBO):

The idea of this metrics is that an object is coupled to another object if two object act upon each other. A class is coupled with another if the methods of one class use the methods or attributes of the other class. An increase of CBO indicates the reusability of a class will decrease. Thus, the CBO values for each class should be kept as low as possible. CBO metric measure the required effort to test the class.
In our proposed method, the entanglement of objects and classes will be reduced by finding related modules (business and data access) and moving them to a separate layer. So that the business layers and data access (according to the layered architecture) are minimally coupled.

Response for a Class (RFC):

RFC is the number of methods that can be invoked in response to a message in a class. According to Pressman [46], since RFC increases, the effort required for testing also increases because the test sequence grows. If RFC increases, the overall design complexity of the class increases and becomes hard to understand. On the other hand, lower values indicate greater polymorphism.
In our proposed method, it is obvious that due to the separation of layers, the number of requests to one layer (compared to the monolithic architecture where all requests reach one layer) will be significantly reduced. The result is that less effort is required in the testing phase.

Lack of Cohesion in Methods (LCOM):

This metric indicates the degree of dependence of the members of a class. Obviously, the higher the degree, the better the classroom design. In the proposed method, by separating the layers, this goal, which is one of the goals of the layered architecture, will be achieved.

5.4. *Performance Evaluation*

Another outcome of the new approach is the performance improvement of the system which is realized with the distributed-ness of the software. This addresses the research question defined at the beginning of the experiment. With the separation of layers and distributing them over nodes, the response time decreases and in result the performance increases. Achieved results in our study also support the previous efforts (e.g. [47]) in which the general assessment of the layered architecture was realized, and the effectiveness of this architecture was approved.
In this experiment, we have used a case study that is related to an educational system in which business and data access layers are implemented monolithically. The results of the experiment show a significant improvement in efficiency after separation of the layers. These results can be generalized to any other more complex system, as the proposed scanner is not limited to a specific pattern, but can accept the rules of any other system as input and recognize and distinguish layers based on them.
In this section, the generated code using the proposed approach is analyzed. To this end, we consider the research variables as follows:
Dependent variable: average response time
Independent variables: connection time, number of layers, execution time of business codes, execution time of data access codes, number of requests (clients), time slice

- BLtrans: BL's transaction time
- DALtrans: DAL's transaction time
- tConnection: Connection time of two nodes
- nClients: Input rate or number of clients
- ts: the size of time slice

First, we consider the original code (without separation) where there is a single layer in the software architecture including DAL and BL layers. In this case, when a request arrives to this blended layer, the response time or return time for this single request will be approximately the sum of the process time for both layers. So,

$$Response\_Time \approx BLtrans + DALtrans$$

When we consider a higher request rate or increase in the number of clients (nClients = n), the response time will be:

$$Response\_Time \approx n * (BLtrans + DALtrans)$$

As the requests are processed one by one (because there is a single layer to process them), all requests will not finish at the same time. So, this response time is not true for all the requests and the average response time for original code needs to be calculated.

If a CPU scheduler such as RoundRobin is used, clearly the requests will be finished in their entering order and the last request will be finished in the last round using the last time slice (ts). The previous requests will be finished earlier than the last one. If we have n requests, in the minimum case, the request n-1 will be finished one ts earlier than the last one. So, we will have:

Request n: the last request will use the last ts and will finish *n-n=0 ts* earlier
Request n-1: the one before the last request will finish *n-(n-1)=1 ts* before the response time
Request n-2: the second before the last request will finish *n-(n-2)=2 ts* before the response time
. . .
Request 1: the first request will finish *n-(1)= n-1 ts* earlier than response time

So, the difference between return time of each request with the (total) response time is $(n-i)*t_s$ for request i. As result, the average response time can be calculated using the response time and these difference times as in equation (1).

On the other hand, when the separated code is generated with our approach, there are two different layers, DAL and BL, which are loaded on two different nodes. Therefore, a connection time will be added to the response time for a single request. So,

$$Response\_Time \approx BLtrans + DALtarns + tConnection$$

As the separated layers (DAL and BL) are running on two different nodes in parallel, the total response time decreases by half. So, for n clients, the response time includes (n*(BLtrans+DALtrans))/2 in addition to tConnection for each client to connect to the server. As result, in the similar way with equation (1), the average response time for the separated code is equation (2).

$$\frac{\sum_{i=1}^{n}(n*(BLTrans+DALTrans))-((n-i)*ts)}{n} \tag{1}$$

$$\frac{\sum_{i=1}^{n}(\frac{1}{2}*(n*(BLTrans+DALTrans)))-((n-i)*ts)+tConnection}{n} \tag{2}$$

In conclusion, the evaluation conducted in this study is generic and is based on the behavior of the separation algorithm introduced in Section 5. To this end, the following parameters are used for both the single layered and the separated layered system: the number of clients, the time for executing BL and DAL layers, time slice, and connection time. As the output, we calculate the response time of the system. Considering the parameters, the times for executing BL and DAL layers are specific for a code/software, the time slice is specific for the operating system, and the connection time is a specific feature of the communication network. Therefore, these parameters are considered as the constant values for the evaluation. However, the number of clients can vary for a code/software (in both single layer and multi-layer).

To illustrate the impact of the code (or layer) separation, the average response times are calculated based on the number of incoming requests (represented by the number of clients as the size of input) for the original (monolithic or single layer) code and separated code using some arbitrary number of clients. Elapsed times are listed in Table 8. This table includes the calculated times based on the equations 1 and 2, the experimental times for a thin client as well as a fat client.

Table 8. The results of response times (in msec) for both the original and separated code

Number of Clients (Input Rate)	Theoretical		Experimental			
			Thin Client		Fat Client	
	Single Layer	Separated Layers	Single Layer	Separated Layers	Single Layer	Separated Layers
1	200	250	29	11	190	144
10	1955	1027.5	65	12	250	210
15	2930	1515	75	19	730	402
20	3905	2002.5	78	40	1055	811
30	5855	2977.5	96	36	1983	1122
50	9755	4927.5	114	62	3200	1900
75	14630	7365	141	90	3910	2026
100	19505	9802.5	171	121	4561	2640
200	39005	19552.5	294	240	7800	4100
500	97504.7	48802.5	702	609	11254	6730
750	146254.3	73177.5	1000	1116	13200	8514
1000	195004.2	97552.5	1336	1201	22360	11700
Assumptions: BLtrans = 100 msec, DALtrans = 100 msec, ts = 10 msec, tConnection = 50 msec						

The results are used in Figure 6 to make the comparison easier. As it can be seen in Figure 6, in the case of having layer separation, the performance of the system is significantly increased, while the average response time is decreased.

5.5. *Experimental Results*

In this section, an experimental evaluation is realized to assess the performance of the proposed algorithm in a real case study. In this experiment, a part of educational system software is used. This software is programmed in Java and uses an Oracle database to store the data. The software is originally implemented in a single layer (blended layer). Its layers are separated (BL and DAL) using the proposed transformation algorithm, as a new version of the software. Also, the configuration of the platform on which the experiment was conducted is:

- The machine running the Business and Data Access Layers: Java 9, MS-Windows 10

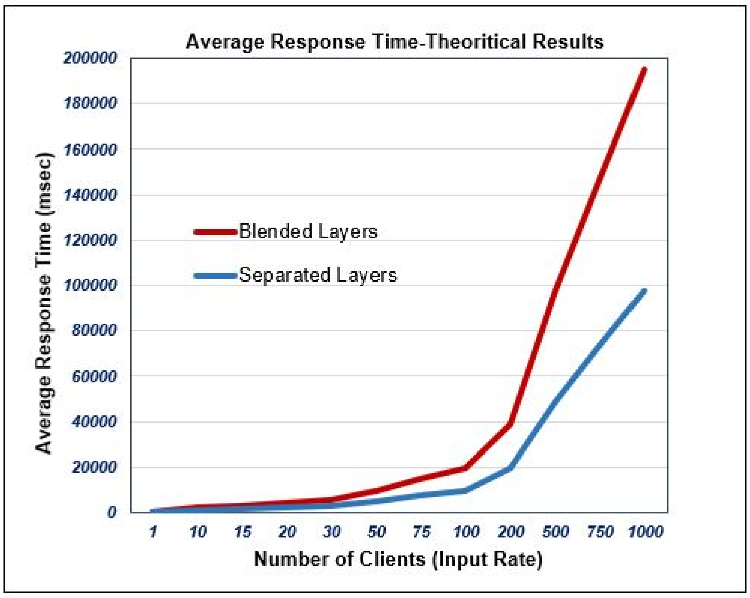


Figure 6. Average response time for both the blended and separated layers based on theoretical conditions

- Server: Windows Server 2016, Oracle 10g
- Network: fast Ethernet, IEEE 802.3u

The two versions of the software are run with 13 different number of clients and the response times are recorded for each of them.

It is worth indicating that the single layer version is run on a single machine while the separated layer version is distributed over two machines (one for each layer).

The entire experiment is repeated twice, one for a thin task with light computation in each client (called thin client) and another for a fat task with heavier computation in each client (called fat client). These experiments exhibit the impact of work load in the result of the proposed transformation algorithm. The results for the experiment with thin clients and fat clients are shown in Figure 7 and Figure 8 respectively.

According to results shown in these diagrams, in the experiment with thin clients, the response times of the separated layer software is slightly better than the single layer software. This is justifiable because separation of small tasks into different layers will not give much gain when they are run in parallel.

In contrary, in the experiment with fat clients, the response times of separated software is far better than ones for the single layer software (due to the parallelizing the big chunks of the main heavy task). This is specially true when the number of clients increases.

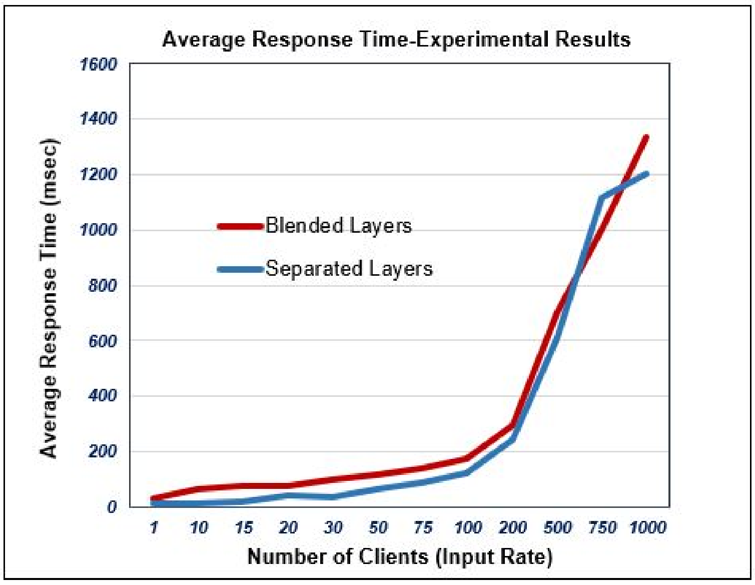


Figure 7. Average response time for both the blended and separated layers based on the experimental results with thin clients

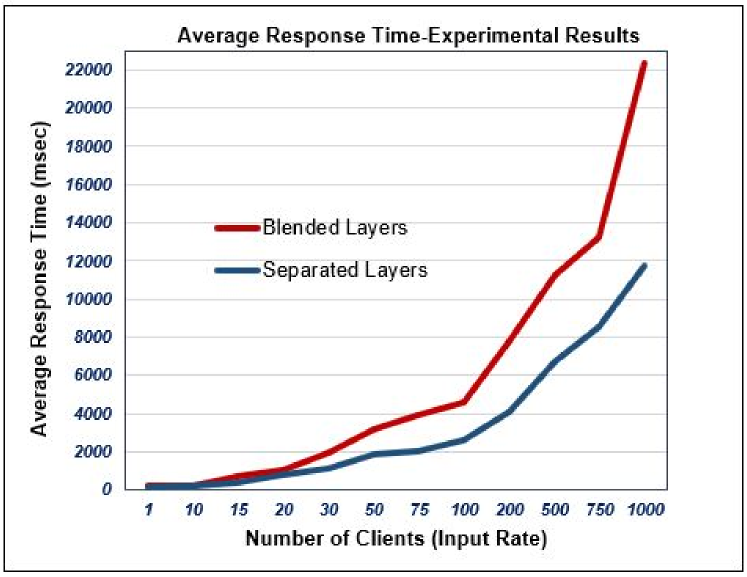


Figure 8. Average response time for both the blended and separated layers based on the experimental results with fat clients

In conclusion, the results of the experiments show that the software with its layers separated using the proposed algorithm has a better performance regarding its response times, as its layers can be run in parallel. The improvement of the performance is noticeable when the clients have heavy computational tasks. This result conforms to the theoretical results previously given in Figure 6. Finally, the achieved results also confirm that applying our method significantly improves the execution of the code and hence answer the research question of the study affirmatively.
A lab-package including the source code of the evaluation case study before and after refactoring as well as all the experiment data is available online for the interested readers at: `https://github.com/alirezakhalilipour/Code-Refactoring-Legacy-to-Layered`.

5.6. *Threats to the Validity*

As it is the case in any experimental study, there can be some risks and threats to the validity of this study. The threats can be originated from different validity types including internal validity, external validity, construction validity, and conclusion/statistical validity [44]. Internal validity relates to a causal relationship between treatment and the outcome. External validity concerns the ability to generalize the results of the study. Construct validity refers to what extent the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions. Conclusion/statistical validity is concerned with the relationship between the treatment and the outcome. In fact, it is the degree to which the relationship of conclusions and data are reasonable.
According to the independent and dependent variables that we introduced in the design subsection, some threats may exist in this study. In the proposed approach, the tConnection, BLTrans, and DALTrans times, used in the formal calculations, are independent variables of the study and Response/Return time is the dependent variable.
Considering the internal threats to the validity, if the tConnection is not in the reasonable range comparing to BLTrans and DALTrans times, the response time can be increased. In other words, if tConnection $<$ BLTrans+DALTrans is not true, the vertical distribution of the layers is not beneficial anymore. Of course, this condition is not usual, but it is possible. Also, even under this condition, the proposed approach still benefits from the other advantages such as OO metrics.
For the threats to external validity, generalization of the achieved results should be considered. In our case study, the input code for the scanner uses JDBC technology to work with data. A possible threat is the scanner is limited with this technology. If the threat becomes true, the scanner and the code transformer cannot be used for the other systems. However, this threat is not serious, as all the other data access models such as ADO.NET, ADO, DAO, ODBC, and RDBC use a similar approach (with JDBC) to handle data and also, they have similar steps to work with data [6]. The only difference between them is the patterns of their instructions. To mitigate

this threat these patterns can be introduced to the scanner to scan the new types of code with the new patterns (but with the same structure and procedure) and generate the separated code. Finally, the patterns are intended as scanner inputs and the scanner is customizable for any new pattern. Therefore, the designed scanner is not limited to a specific pattern and can be customized for other patterns.
Regarding the construct validity, this study uses the response time measures of the system for evaluating the performance of the generated multi-layer architecture comparing to the monolithic structure. A threat seems to be not considering the turn-around time of the data access technology such as JDBC and the data base, such as Oracle. These times may have impact in the total performance of the system. However, to mitigate this threat, we have considered these times as part of the total response time. As we have used the same setup for both monolithic and multi-layer systems, the data access technology and the database used in this study have the same impact in both scenarios. As a result, this threat has no affect in evaluating our approach.
Finally, for the conclusion threat to the validity, we have assessed some results from the data collected by the experiment. Furthermore, these results were already validated using the formulas developed for the response time. Therefore, the relation between the collected data and the conclusions are reasonable, based on the mathematical formulas and this resolves the conclusion threat.

6. Conclusion and Future Work

Most of the information systems were designed with two or three-layer software architectures in the previous two decades. In these architectures, codes for Process/Logic and DALs locate in a single layer called BL which leads to a bottleneck or system crash when the process load or number of the user increases. One possible approach to tackle with this problem is re-designing the systems with a new architecture and separating these layers. However, this solution is very costly, with the cost of developing the system from scratch. Another approach is to modify the available system and separate the layers. Although this approach is less costly, it is complex and time consuming when applied manually, due to the blended codes of two layers in a single layer. To address these problems, a solution can be the use of compiler techniques and make the process of this separation automatic, in a way that a tool can detect the bottleneck points and resolve them automatically. In this way, it can be possible to separate the codes for Process/Logic and DALs in the legacy information systems with classical two or three layer architectures.

In this regard, the separation of software layers in an information system has been discussed in this paper and a code scanner has been introduced which can scan the middle layer of the legacy/traditional software and make some modifications on the parts of the program which access the data to pave the way for the separation of BL and DAL. The initial codes, a combination of the data access and the business process codes, are separated automatically and placed in two different layers. In this way, it is possible to run these separated layers in different nodes and balance the processing load. This approach brings the following advantages: the possibility of rapid modifications on software, the increase in the development speed, the increase of the performance and the scalability, and easier software maintenance. According to the quantitative evaluation results, the average response time can decrease to half, especially for the larger number of clients.

Considering the execution time of the separated layers, the only cost which is brought by adding the new layer is the time of function/method calls between layers. Since the code in the new layer comes from the original code, the additional time required for these calls is only the time needed for the connection between two layers. If the layers generated by the scanner are placed in the same node, the calls will be local, and the connection time would be trivial. But if the layers are located on different nodes, the connection time will be longer (due to a remote call). However, as the BL and the DAL run in parallel, the connection time is ignorable considering the time of running the layers. Taking into account the memory consumption, the only additional memory would be the creation of a few objects to make the connection between two layers. The number and the size of these objects are very limited, and they are destructed as soon as the call is realized. So, the new approach will not bring a significant memory overhead.

As a future work, the remote procedure calls can be transformed from synchronous to asynchronous to increase the performance. In this way, after a method call, the client will not wait for the termination of the method and can continue its task, if it does not need for the result of the method. So, the system will not waste time waiting for termination of a method; instead, the Business codes can be executed in concurrent with Data Access codes. This will lead to the parallel execution of different parts of the application on different nodes which increases the performance of the system.

As another possible study, the separation mechanism can be performed with the use of domain-specific languages and model transformations as we experienced (e.g. in [48], [49], [50], [51]) for software agent architecture models. In this way the model of the layered architecture can be generated and re-used in later extensions. Also, the high-level validation and verification can be performed on the system model [52].

Acknowledgments

Mehmet Onat, Hale Gezgen and Geylani Kardas are funded by KocSistem Information and Communication Services Inc. and the Scientific and Technological Research Council of Turkey (TUBITAK) under European Union EUREKA ITEA labelled (ITEA project no: 14014) R&D project ASSUME (TUBITAK Grant no: 9150181). Alireza Khalilipour and Moharram Challenger would also like to thank KocSistem Information and Communication Services Inc for their funding and supporting the study. Finally, Alireza Khalilipour would like to thank "Sama Technical and

Vocational Training College" for their support during this study.

Bibliography

[1] A. Sharma, M. Kumar and S. Agarwal, A complete survey on software architectural styles and patterns, *Procedia Computer Science* **70** (2015) 16–28.

[2] M. Challenger, F. Erata, M. Onat, H. Gezgen and G. Kardas, A model-driven engineering technique for developing composite content applications, in *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik2016.

[3] C. Pahl, P. Jamshidi and O. Zimmermann, Architectural principles for cloud software, *ACM Transactions on Internet Technology (TOIT)* **18**(2) (2018) p. 17.

[4] R. Wang, S. Ying and X. Jia, Log Data Modeling and Acquisition in Supporting SaaS Software Performance Issue Diagnosis, *International Journal of Software Engineering and Knowledge Engineering* **29**(9) (2019) 1245–1277.

[5] O. Vogel, I. Arnold, A. Chughtai and T. Kehrer, *Software architecture: a comprehensive framework and guide for practitioners* (Springer Science & Business Media, 2011).

[6] A. A. B. Baqais and M. Alshayeb, Automatic software refactoring: a systematic literature review, *Software Quality Journal* (2019) 1–44.

[7] Y. Wang, H. Yu, Z. Zhu, W. Zhang and Y. Zhao, Automatic software refactoring via weighted clustering in method-level networks, *IEEE Transactions on Software Engineering* **44**(3) (2017) 202–236.

[8] H. K. A. Bakar, R. Razali and D. I. Jambari, Implementation phases in modernisation of legacy systems, in *2019 6th International Conference on Research and Innovation in Information Systems (ICRIIS)*, IEEE2019, pp. 1–6.

[9] J. Kazanavičius and D. Mažeika, Migrating legacy software to microservices architecture, in *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, IEEE2019, pp. 1–5.

[10] R. Chen, S. Li and Z. Li, From monolith to microservices: a dataflow-driven approach, in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE2017, pp. 466–475.

[11] L. Carvalho, A. Garcia, W. K. Assunção, R. Bonifácio, L. P. Tizzei and T. E. Colanzi, Extraction of configurable and reusable microservices from legacy systems: An exploratory study, in *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, 2019, pp. 26–31.

[12] W. K. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio and A. Egyed, Reengineering legacy applications into software product lines: a systematic mapping, *Empirical Software Engineering* **22**(6) (2017) 2972–3016.

[13] M. Wyrich and J. Bogner, Towards an autonomous bot for automatic source code refactoring, in *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, IEEE2019, pp. 24–28.

[14] M. Richards, *Software architecture patterns* (O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA . . . , 2015).

[15] R. Grycuk, M. Gabryel, R. Scherer and S. Voloshynovskiy, Multi-layer architecture for storing visual data based on wcf and microsoft sql server database, in *International Conference on Artificial Intelligence and Soft Computing*, Springer2015, pp. 715–726.

[16] U. K. Roy, *Advanced Java Programming* (Oxford University Press India, 2015).

[17] A. S. Ganesan and T. Chithralekha, A survey on survey of migration of legacy systems, in *Proceedings of the International Conference on Informatics and Analytics*, *ICIA-16*, (ACM, New York, NY, USA, 2016), pp. 72:1–72:10.

[18] Á. Vathy-Fogarassy and T. Hugyák, Uniform data access platform for sql and nosql database systems, *Information Systems* **69** (2017) 93–105.

[19] S. Parsa and O. Bushehrian, Performance-driven object-oriented program remodularisation, *IET software* **2**(4) (2008) 362–378.

[20] M. F. Dolz, D. D. R. Astorga, J. Fernández, J. D. García and J. Carretero, Towards automatic parallelization of stream processing applications, *IEEE Access* **6** (2018) 39944–39961.

[21] K. Alsubhi, F. Alsolami, A. Algarni, K. Jambi, F. Eassa and M. Khemakhem, An architecture for translating sequential code to parallel, in *Proceedings of the 2nd International Conference on Information System and Data Mining*, ACM2018, pp. 88–92.

[22] S. Parsa and O. Bushehrian, Automatic translation of serial to distributed code using corba event channels, in *International Symposium on Computer and Information Sciences*, Springer2005, pp. 152–161.

[23] S. Muhammad, O. Maqbool and A. Q. Abbasi, Evaluating relationship categories for clustering object-oriented software systems, *IET software* **6**(3) (2012) 260–274.

[24] M. Alkhalid, M. Alshayeb and S. A. Mahmoud, Software refactoring at the package level using clustering techniques, *Software* **5**(3) (2011) 276–284.

[25] R. C. Millham, Data reengineering of legacy systems, in *Enterprise Information Systems: Concepts, Methodologies, Tools and Applications*, (IGI Global, 2011) pp. 181–188.

[26] B. Andreopoulos, A. An, V. Tzerpos and X. Wang, Multiple layer clustering of large software systems, in *12th Working Conference on Reverse Engineering (WCRE'05)*, IEEE2005, pp. 10–pp.

[27] S. Bobde and R. Phalnikar, Restructuring of object-oriented software system using clustering techniques, in *Proceeding of International Conference on Computational Science and Applications*, Springer2020, pp. 419–425.

[28] B. M. Santos, I. G.-R. de Guzmán, V. V. de Camargo, M. Piattini and C. Ebert, Software refactoring for system modernization, *IEEE Software* **35**(6) (2018) 62–67.

[29] M. Wahler, U. Drofenik and W. Snipes, Improving code maintainability: A case study on the impact of refactoring, in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE2016, pp. 493–501.

[30] O. Zimmermann, Architectural refactoring: A task-centric view on software evolution, *IEEE Software* **32**(2) (2015) 26–29.

[31] P. Kandukuri, Software modernization through model transformations, in *First International Conference on Artificial Intelligence and Cognitive Computing*, Springer2019, pp. 165–174.

[32] U. Zdun, M. Voelter and M. Kircher, Design and implementation of an asynchronous invocation framework for web services, in *International Conference on Web Services*, Springer2003, pp. 64–78.

[33] A. Khalilipour and M. Challenger, Automatic conversion of remote invocations in optimization of distributed codes, *Journal of Academic and Applied Studies* **2**(4) (2012) 22–33.

[34] A. Khalilipour and M. Challenger, An event-based approach on automatic synchronous-to-asynchronous transformation of web service invocations, in *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*, IEEE2019, pp. 162–169.

[35] M. Chavan, R. Guravannavar, K. Ramachandra and S. Sudarshan, Program transformations for asynchronous query submission, in *2011 IEEE 27th International Conference on Data Engineering*, IEEE2011, pp. 375–386.

[36] B. Baliś, M. Bubak and M. Wegiel, A solution for adapting legacy code as web services, in *Component Models and Systems for Grid Applications*, (Springer, 2005) pp. 57–75.
[37] R. Khadka, A. Saeidi, S. Jansen, J. Hage and G. P. Haas, Migrating a large scale legacy application to soa: Challenges and lessons learned, in *2013 20th Working Conference on Reverse Engineering (WCRE)*, IEEE2013, pp. 425–432.
[38] M. A. Al Sheikh, H. A. Aboalsamh and A. Albarrak, Migration of legacy applications and services to service-oriented architecture (soa), in *The 2011 International Conference and Workshop on Current Trends in Information Technology (CTIT 11)*, IEEE2011, pp. 137–142.
[39] P. Bjeljac, B. Perišić, I. Zečević and D. Venus, Refactoring legacy enterprise information systems to service oriented architecture-the faculty of technical sciences case study (2014).
[40] H. M. Sneed *et al.*, Wrapping legacy software for reuse in a soa, in *Multikonferenz Wirtschaftsinformatik*, **2**2006, pp. 345–360.
[41] K. Avila, P. Sanmartin, D. Jabba and M. Jimeno, Applications based on service-oriented architecture (soa) in the field of home healthcare, *Sensors* **17**(8) (2017).
[42] Ó. M. Pereira, R. L. Aguiar and M. Y. Santos, Reusable business tier architecture driven by a wide typed service, in *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*, IEEE2013, pp. 135–141.
[43] R. Heckel, R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos and L. Andrade, Architectural transformations: From legacy to three-tier and services, in *Software Evolution*, (Springer, 2008) pp. 139–170.
[44] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in software engineering* (Springer Science & Business Media, 2012).
[45] C. Jones, *Applied software measurement: global analysis of productivity and quality* (McGraw-Hill Education Group, 2008).
[46] R. S. Pressman, *Software engineering: a practitioner's approach* (McGraw-Hill Education, 2019).
[47] S. Duttagupta and M. Nambiar, Maximum throughput computation of an application in a multi-tier environment, in *2012 International Symposium on Performance Evaluation of Computer & Telecommunication Systems (SPECTS)*, IEEE2012, pp. 1–7.
[48] M. Challenger, S. Demirkol, S. Getir, M. Mernik, G. Kardas and T. Kosar, On the use of a domain-specific modeling language in the development of multiagent systems, *Engineering Applications of Artificial Intelligence* **28** (2014) 111–141.
[49] G. Kardas, E. Bircan and M. Challenger, Supporting the platform extensibility for the model-driven development of agent systems by the interoperability between domain-specific modeling languages of multi-agent systems, *Computer Science and Information Systems* **14**(3) (2017) 875–912.
[50] G. Kardas, Z. Demirezen and M. Challenger, Towards a DSML for semantic web enabled multi-agent systems, in *Proceedings of the International Workshop on Formalization of Modeling Languages*, 2010, pp. 1–5.
[51] M. Challenger, B. Tezel, O. Alaca, B. Tekinerdogan and G. Kardas, Development of semantic web-enabled BDI multi-agent systems using SEA_ML: An electronic bartering case study, *Applied Sciences* **8**(5) (2018) p. 688.
[52] M. Challenger, G. Kardas and B. Tekinerdogan, A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems, *Software Quality Journal* **24**(3) (2016) 755–795.