

# AUTOMATED PROCEDURE CLUSTERING FOR REVERSE ENGINEERING PL/SQL PROGRAMS

A Thesis

by

Metin Altınışık

Submitted to the  
Graduate School of Sciences and Engineering  
In Partial Fulfillment of the Requirements for  
the Degree of

Master of Science

in the  
Department of Computer Science

Özyeğin University  
May 2016

Copyright © 2016 by Metin Altınışık

# AUTOMATED PROCEDURE CLUSTERING FOR REVERSE ENGINEERING PL/SQL PROGRAMS



Approved by:

---

Asst. Prof. Hasan Sözer (Advisor)  
Department of Computer Science  
*Özyeğin University*

---

Asst. Prof. Mehmet Aktaş  
Department of Computer Engineering  
*Yıldız Technical University*

---

Asst. Prof. Gonca Gürsun (Advisor)  
Department of Computer Science  
*Özyeğin University*

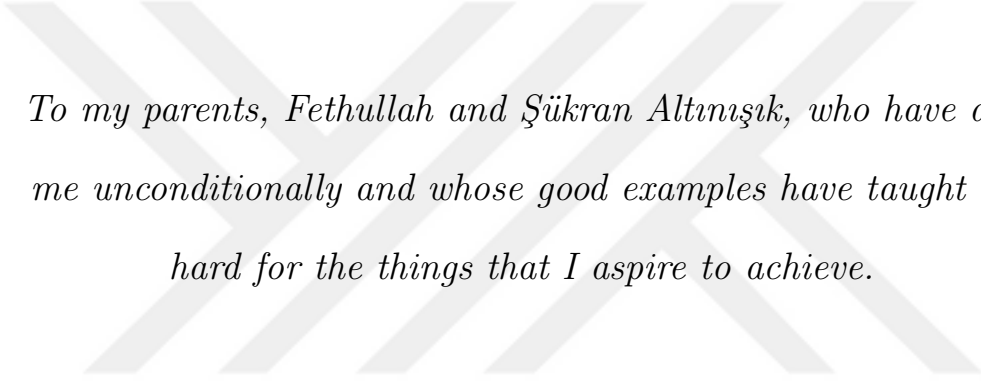
---

Assoc. Prof. Ali Fuat Alkaya  
Computer Science and Engineering  
Department  
*Marmara University*

Date Approved: ... ..... 2016

---

Asst. Prof. Barış Aktemur  
Department of Computer Science  
*Özyeğin University*



*To my parents, Fethullah and Şükran Altınışik, who have always loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.*

*This thesis work is also dedicated to my son Denizhan, my daughter Ebru İpek and my wife, Nagihan, who has been a constant source of support and encouragement during the challenges of graduate school and life. I am truly thankful for having them in my life.*

## ABSTRACT

Large software systems have to be decomposed into separate, modular units for providing appropriate abstractions and improving maintainability. There exist clustering techniques that are applied to provide such abstractions by automatically grouping system modules based on dependencies among them. Hereby, dependency is usually measured as the extent to which a module refers to elements of another module. This approach cannot be directly applied for all types of programs. Some programs involve modules that are indirectly coupled. For instance, PL/SQL programs include procedures that are in most cases coupled due to their database operations although they do not make calls to each other. In this thesis, we provide an approach and a tool that supports automated modularization of software systems by considering this type of dependencies. We also extend this approach for multiple, different types of dependencies. We construct several dependency matrices each of which captures a different type of dependency among the system modules. First, we perform clustering according to each of these matrices separately. Then, we perform cluster aggregation (meta clustering) on the obtained clustering results to propose a packaging structure to the designer. We performed two industrial case studies on real PL/SQL programs from the telecommunications domain. Many unexisted packages were proposed by our tool and the accuracy of the results were confirmed by domain experts.

## ÖZETÇE

Büyük yazılım sistemlerinin bakımlarını daha verimli ve kaliteli bir şekilde gerçekleştirebilmek için bu sistemleri modüller ve daha küçük birimlere ayırmak bir zorunluluktur. Bu şekilde bir modüler yapı kurmak için bu tür sistemlerin aralarındaki bağımlılıklarına göre otomatik olarak gruplamak için mevcutta değişik gruplama teknikleri mevcuttur. Burada, bir modülün diğerine bağımlılığı genellikle bu modülün diğer modülün bileşenlerine bağımlılığı yönünden ölçülür. Bu yaklaşım tüm programlar için maalesef direk olarak uygulanamamaktadır. Çünkü bazı programlar dolaylı olarak birbirlerine bağımlı modüller içerebilirler. Örneğin, PL/SQL programları birbirlerini çağırmadıkları halde aynı veritabanı operasyonları yönünden birbirlerine oldukça bağımlı olan ayırık prosedürler içerirler. Bu tezde, bu tipteki bağımlılıkları dikkate alarak sistemleri modüler bir yapıya otomatik dönüştürebilecek bir yaklaşım ve araç geliştirdik. Ayrıca bu yaklaşımı farklı tipteki bağımlılıkları içerecek şekilde genişleterek, sistem modülleri arasındaki farklı tipte bağımlılıkları içeren bağımlık matrisleri oluşturduk. İlk olarak, her bir matris için ayrı ayrı olarak gruplama çalışması yaptık. Sonrasında, ilk çalışmadan çıkan gruplamaları kümüle bir gruplama mekanizmasına dahil ederek nihai gruplamaları elde ettik. Çıkan sonuçları mevcut modülleri bir paketleme önerisi olarak geliştiricilere sunduk. Bu çalışmayı telekomunikasyon alanında bulunan iki farklı büyük ve kompleks sistem için (CRM ve Faturalama) ayrı ayrı uyguladık. Bir çok ayırık prosedür geliştirdiğimiz bu araç ile gruplanabildiğini gördük. Ayrıca, çıkan sonuçları yazılım mimarları tarafından doğruluğunu teyit ettirdik.

## ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor, Dr. Hasan Sözer for all his help and guidance that he has given me over the past two years. Secondly, I would also like to thank Dr. Gonca Gürsun for providing me support during this period. Finally, I would like to thank software developers at Turkcell Technology for sharing their code base with me and supporting the my case study.

# TABLE OF CONTENTS

<b>DEDICATION</b> . . . . .	<b>iii</b>
<b>ABSTRACT</b> . . . . .	<b>iv</b>
<b>ÖZETÇE</b> . . . . .	<b>v</b>
<b>ACKNOWLEDGMENTS</b> . . . . .	<b>vi</b>
<b>LIST OF TABLES</b> . . . . .	<b>ix</b>
<b>LIST OF FIGURES</b> . . . . .	<b>x</b>
<b>I INTRODUCTION</b> . . . . .	<b>1</b>
<b>II BACKGROUND</b> . . . . .	<b>4</b>
2.1 Design Structure Matrix . . . . .	4
2.2 Cluster Aggregation . . . . .	6
2.3 PL/SQL Programs . . . . .	7
<b>III RELATED WORK</b> . . . . .	<b>11</b>
<b>IV OVERALL APPROACH</b> . . . . .	<b>14</b>
4.1 Single DSM Approach (SDSM) . . . . .	14
4.2 Multi DSM Approach (MDSM) . . . . .	16
<b>V INDUSTRIAL CASE STUDY</b> . . . . .	<b>22</b>
5.1 Legacy Systems . . . . .	22
5.2 Case Studies . . . . .	23
<b>VI RESULTS AND DISCUSSION</b> . . . . .	<b>27</b>
6.1 Results . . . . .	27
6.2 Discussions . . . . .	29
6.3 Threats to Validity . . . . .	30
<b>VII CONCLUSIONS AND FUTURE WORK</b> . . . . .	<b>32</b>
<b>APPENDIX A — GENERATED DSM MODELS FOR CRM</b> . . . . .	<b>34</b>

APPENDIX B — GENERATED DSM MODELS FOR BILLING 37  
REFERENCES . . . . . 41





## LIST OF TABLES

1	An example of clustering aggregation . . . . .	7
2	Support Vector Clustering parameters . . . . .	24
3	<i>SDSM Cluster Analyzer</i> results for CRM System . . . . .	27
4	<i>SDSM Cluster Analyzer</i> results for Billing System . . . . .	28
5	<i>MDSM Cluster Analyzer</i> results for CRM System . . . . .	29
6	<i>MDSM Cluster Analyzer</i> results for Billing System . . . . .	30



## LIST OF FIGURES

1	A sample DSM with 4 procedures. . . . .	4
2	The sample DSM in Figure 1 after a reordering of rows and columns. . . . .	5
3	Packages are used for organizing PL/SQL programs to manage complexity. . . . .	10
4	The overall SDSM approach. . . . .	15
5	The overall MDSM approach. . . . .	17
6	A snippet from the generated DSM model, capturing the set of database elements that are commonly accessed by system modules for CRM system. . . . .	26
7	A snippet from the generated DSM model, capturing model interdependency in terms of direct references in the source code for the CRM system. (The rows and columns are reordered to highlight cells that have the value 1 in the sparse matrix) . . . . .	35
8	A snippet from the generated DSM model, capturing modifications to system modules that are applied by the same developer at the same time for the CRM system. (The rows and columns are reordered to highlight cells that have the value greater than 0 in the matrix) . . . . .	36
9	A snippet from the generated DSM model, capturing the set of database elements that are commonly accessed by system modules for the Billing system. . . . .	38
10	A snippet from the generated DSM model, capturing model interdependency in terms of direct references in the source code for the Billing system. (The rows and columns are reordered to highlight cells that have the value 1 in the sparse matrix) . . . . .	39
11	A snippet from the generated DSM model, capturing modifications to system modules that are applied by the same developer at the same time for the Billing system. (The rows and columns are reordered to highlight cells that have the value greater than 0 in the matrix) . . . . .	40

# CHAPTER I

## INTRODUCTION

Modularity is one of the basic principles for supporting maintainability of software systems [1]. Especially, large scale software systems have to be decomposed into separate, modular units for providing appropriate abstractions. This decomposition plays an important role for software architecture design, which embodies key design decisions, gross-level components of a system, and their interactions [2, 3]. As such, gross-level software decomposition is described as part of the software architecture documentation, which is an important artifact for maintaining an evolving system [4].

Software architecture documentation might be unavailable for legacy systems. Even if there exist documentation, this documentation can turn out to be obsolete. It can be inconsistent with the actual implementation of the system. Inconsistencies might arise due to the evolution of the software system without necessary updates being applied to documentation. This can lead to *architectural drift* [5, 6], which is defined as the introduction of unintended design decisions that create architectural anomalies.

Reverse engineering [7] and in particular, software architecture reconstruction [8] approaches have been introduced to recover incorrect or incomplete architectural documentation for a software system. Many of these approaches apply clustering techniques on software modules to group them and as such, infer the high-level structure of the system. Clustering is performed according to inter-module dependencies that are mainly identified with static and/or dynamic analysis on the source code. Hereby, dependency is usually measured as the extent to which a module refers to elements of another module. Most of the approaches [8] consider direct dependencies

(e.g., function call, variable access, etc.) [9, 10] only. There also exist approaches that analyze information flow among the modules [11], their involvement in common usage scenarios [12] and lexical similarities among comments in their source code [11]. In this work, we focused on application programs that are developed with the PL/SQL language. These programs include procedures that are in most cases coupled due to their database operations although they do not necessarily make calls to each other. We introduce and evaluate two different approaches for clustering PL/SQL programs by explicitly modeling this type of dependencies among the procedures.

In our first approach, we derive (indirect) dependencies among PL/SQL procedures by analyzing the accessed database elements. We represent these dependencies in the form of a design structure matrix (DSM). Then, we cluster the procedures and propose a packaging structure to the designer. An initial evaluation of this approach showed promising results in which most of the procedures were successfully clustered in relevant packages. An analysis of the remaining procedures revealed that some of them directly refer to each other or they use common resources other than database elements. Therefore, we introduced a second approach that utilizes input regarding multiple types of dependencies at the same time.

In our second approach, we construct multiple DSMs, each of which captures a different type of dependency among the system modules. First, we perform clustering according to each of these DSMs separately. Then, we perform cluster aggregation (meta clustering) on the obtained clustering results to propose a packaging structure to the designer. We performed two industrial case studies on real PL/SQL programs from the telecommunications domain. We employed 3 types of dependencies based on: *i*) access to common database elements, *ii*) calls to common procedures, and *iii*) modifications by common software developers at the same time. These dependency types are aligned with the categorization provided before [13]. An analysis of the results showed that our approaches successfully clustered a large set of stand-alone

procedures that did not belong to any package. The accuracy of these results were confirmed by domain experts.

The remainder of this thesis is organized as follows. In the following chapter, we provide background information on design structure matrices, cluster aggregation and PL/SQL programs. We summarize the related studies in Chapter 3. We present the approach in Chapter 4, which is illustrated in Chapter 5, in the context of the industrial case study. We discuss the results in Chapter 6. Finally, in Chapter 7, we provide the conclusions and discuss for possible future work directions.

## CHAPTER II

### BACKGROUND

In this chapter, we provide background information on design structure matrices, cluster aggregation and PL/SQL programs.

#### 2.1 *Design Structure Matrix*

*Design structure matrix (DSM)* is a modeling technique for managing complex systems [14]. It provides a compact, visual, intuitive representation. DSM is utilized in many domains including engineering management, finance and social sciences. It has also been applied for reasoning about software architectures [15, 16]. In this context, it is also known as the *dependency structure matrix* [17].

	P1	P2	P3	P4
P1			15	20
P2			5	
P3	30			40
P4				

**Figure 1:** A sample DSM with 4 procedures.

Figure 1 depicts a DSM for 4 modules, enumerated as P1, P2, P3 and P4. Hereby, the rows and columns of the DSM represent the same modules. Each cell on the diagonal represents the dependency of a module to itself. These cells are not relevant and as such, they are shaded. The other cells represent the amount of dependencies between different pairs of modules. For instance, we can see that the number of dependencies from P3 to P1 is 15. An empty cell represents a lack of dependency

between the corresponding modules. For example, we can see that P2 does not depend on any other module.

	P4	P1	P3	P2
P4				
P1	20		15	
P3	40	30		
P2			5	

**Figure 2:** The sample DSM in Figure 1 after a reordering of rows and columns.

The columns and rows of a DSM are usually reordered to reveal highly coupled modules of a system. The reordering can start with modules that depend on most of the other modules and end with modules that are mostly depended by other modules [16]. Figure 7 shows the result of such a reordering strategy for the DSM from Figure 1. The reordering yields that P1 and P3 are highly coupled with each other. Hence, they can be grouped to form a package.

DSM can help to reason about how coupled the modules of a system are. Different clustering algorithms can be used for restructuring a DSM and for grouping highly coupled modules together. This grouping can point out the inherent structure of the system that supports the reverse engineering of its software architecture. Results can also point out a need for refactoring to better modularize the system.

In this work, we employ DSMs for reasoning about PL/SQL programs. We consider a PL/SQL procedure as the unit of a module. In section 2.3 , we shortly introduce PL/SQL and its distinctive features with respect to other programming languages.

## 2.2 Cluster Aggregation

Cluster aggregation is an approach to clustering that is based on the concept aggregation [18]. It takes a set of clusterings as input and aims at finding a single clustering that agrees as much as possible with them. It can be used as a meta-clustering method to increase the accuracy of a set of existing clusterings [19] [20].

Assume that we have set of objects and we have information about their clustering information. This clustering information comes in the form of  $n$  clusterings  $K_1, \dots, K_n$ . The aim of cluster aggregation is to find a single clustering  $K$  that agrees as much as possible with the  $n$  clusterings. Hereby, a disagreement is defined between two clusterings  $K$  and  $K'$  as a pair of objects  $(x, y)$  such that clustering  $K$  places them in the same cluster, while clustering  $K'$  places them in different clusters, or vice versa. If  $f(K, K')$  represents the number of disagreements between  $K$  and  $K'$ , then the aim is to find a clustering  $K$  that minimizes  $\sum_{i=1}^n f(K_i, K)$ .

As an example, consider the dataset provided as

$$D = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$$

that consists of eight objects, and assume that we have four clustering information for  $D$  like below;

$$K_1 = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}, \{x_7, x_8\}\}$$

$$K_2 = \{\{x_1, x_3\}; \{x_2, x_4\}; \{x_5\}, \{x_6\}, \{x_7, x_8\}\}$$

$$K_3 = \{\{x_1, x_3\}, \{x_2, x_4\}, \{x_5, x_6\}, \{x_7, x_8\}\}$$

$$K_4 = \{\{x_1, x_3\}, \{x_2, x_4\}, \{x_5, x_6\}, \{x_7\}, \{x_8\}\}$$

Table 1 shows the 4 clusterings, where each column represents a clustering, and a value  $i$  in each cell shows that the tuple in that row belongs to the  $i^{th}$  cluster of the clustering in that column.



The rightmost column is the clustering  $K = \{\{x_1, x_3\}, \{x_2, x_4\}, \{x_5, x_6\}, \{x_7, x_8\}\}$  that minimizes the total number of disagreements with the clusterings  $K_1, K_2, K_3$ , and  $K_4$ . In this example, the total number of disagreements is 6: one with the clustering  $K_2$  for the pair  $(x_5, x_6)$ , one with the clustering  $K_4$  for the pair  $(x_7, x_8)$ , and 4 with the clustering  $K_1$  for the pairs  $(x_1, x_2), (x_1, x_3), (x_2, x_4)$ , and  $(x_3, x_4)$ .

	$K_1$	$K_2$	$K_3$	$K_4$	$K$
$x_1$	1	1	1	1	1
$x_2$	1	2	2	2	2
$x_3$	2	1	1	1	1
$x_4$	2	2	2	2	2
$x_5$	3	3	3	3	3
$x_6$	3	4	3	3	3
$x_7$	5	5	5	5	5
$x_8$	5	5	5	6	5

**Table 1:** An example of clustering aggregation

In this work, we perform cluster aggregation on different clusterings of procedures in PL/SQL programs. In the following, basic properties of these programs are explained.

### 2.3 PL/SQL Programs

Structural Query Language (SQL) is used for performing basic operations on a database such as *select*, *insert*, *delete* and *update*; however, its declarative structure and expressiveness fall short for developing large applications. PL/SQL (Procedural Language/Structured Query Language) is a 3<sup>rd</sup> generation language that combines procedural language features with SQL such that SQL statements can be intermixed with imperative code [21]. PL/SQL programs directly run on a Oracle<sup>1</sup> database management system without having to establish a separate connection. Its hybrid nature makes it easy to develop large and complex applications. Significant part of enterprise applications today are developed with PL/SQL.

---

<sup>1</sup>www.oracle.com

A PL/SQL program is composed of procedures and functions that can be grouped into packages. The basic difference between procedures and function is that while functions return a value procedures do not return a value. There can also be standalone procedures and functions, which do not belong to any package. All the procedures and functions are compiled and stored as part of the database. After being compiled and stored in the database, they can be called by any application which connects to the database.

**Listing 2.1:** The general structure of a PL/SQL block.

```
1 PROCEDURE P(id IN NUMBER) IS
2   sales NUMBER;
3   total NUMBER;
4   ratio NUMBER;
5 BEGIN
6   SELECT x,y INTO sales,total
7     FROM result WHERE result_id = id;
8   ratio := sales/total;
9   IF ratio > 10 THEN
10    INSERT INTO comp VALUES (id,ratio);
11  END IF;
12  COMMIT;
13 EXCEPTION
14  WHEN ZERO_DIVIDE THEN
15    INSERT INTO comp VALUES (id,0);
16    COMMIT;
17  WHEN OTHERS THEN
18    ROLLBACK;
19  END;
```

PL/SQL procedures consist of 3 main parts as listed in Listing 2.1. The declaring part (Lines 1-4) identifies any used variables or constants. The executable part, starts with BEGIN keyword and ends with END keyword, (Lines 5 and 19, respectively)

contains the main logic. An exception-handling part, starts with `EXCEPTION` keyword, (Line 13) handles errors that may be occurred in the executable part of the PL/SQL code. The first two parts are mandatory, whereas the last part is optional.

Enterprise level applications are usually large and complex. Hence, packages are used for organizing and grouping PL/SQL procedures and functions to manage complexity. Packages are composed of two parts: *i)* specification, and *ii)* body. The specification part defines the interface of the package. It declares the types, variables, constants, exceptions, functions and procedures that can be referenced from outside of the package [21]. The form of a standard package specification is shown in Listing 2.2 .

**Listing 2.2:** The structure of a standard package specification in PL/SQL [21].

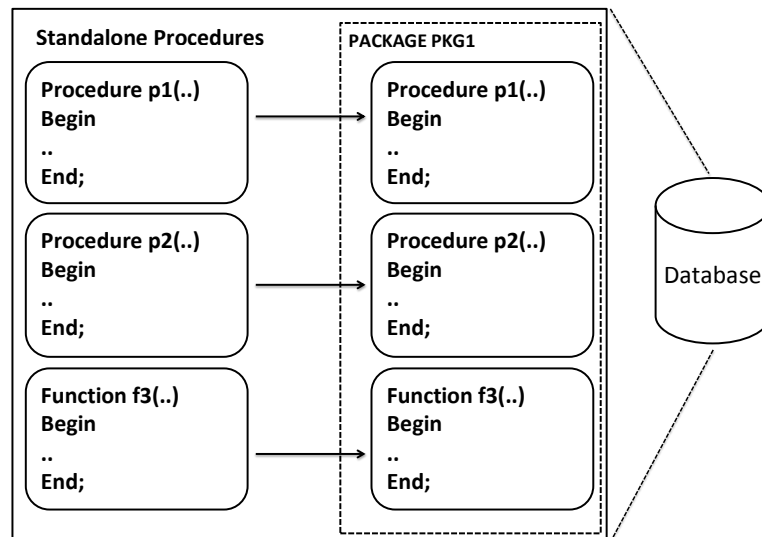
```
1 CREATE OR REPLACE PACKAGE
2   EGPKC
3     variables
4     constants
5     exceptions
6     procedure procedure_a(param1, ...);
7     function function_b(param1,...)   return varchar2;
8 END;
```

The package body contains the implementation of the programs which are declared in the package specification and other private subprograms. The general outline of a package body is illustrated in Listing 2.3. Related standalone procedures/functions can be grouped under a package to manage complexity. Figure 3 shows an example, which involves 2 standalone procedures *p1* and *p2*, and a standalone function named *f3*. Hereby, these 3 standalone objects are placed under a package named *PKG1*.

In the following chapter, we introduce our approach for automatically clustering related procedures to reverse engineer the architecture of the system and to propose a packaging structure to the designer.

**Listing 2.3:** The structure and contents of a package body in PL/SQL [21].

```
1 CREATE OR REPLACE PACKAGE BODY
2   package_name EGPCCK
3   PROCEDURE procedure_1(arg1,...) IS
4     BEGIN
5       ...
6     EXCEPTION ...
7   END procedure_1;
8   FUNCTION function_1(arg1,...)
9     RETURN data_type IS
10      result_variable data_type
11    BEGIN
12      ...
13    RETURN result_variable;
14  EXCEPTION ...
15 END function_1;
16 END package_name;
```



**Figure 3:** Packages are used for organizing PL/SQL programs to manage complexity.

## CHAPTER III

### RELATED WORK

There exist many tools and methods that have been developed for software reverse engineering. Hereby, the goal is to recover incorrect, incomplete or unavailable documentation, and as such to aid in the maintenance of legacy systems [7]. There exists an extensive literature [22, 7, 8] on this subject.

Some of the existing techniques aim at revealing dependencies among the high-level modules/components of a system and visualize these dependencies [23, 24]. Other reverse engineering techniques derive low level models from a program like call graphs or program dependency graphs [25]. Then, complementary tools [26] can apply clustering algorithms on these models to infer main modules of a software system and the inter-dependencies among them. Some of the approaches focus on analyzing the runtime behavior for reconstructing execution scenarios [24] and behavioral views [27]. In another approach, execution traces of a system are utilized to support static analysis by identifying related modules that are involved in the same usage scenarios [12]. There are also tools that combine static and dynamic analysis to construct both structural and behavioral views [23, 28]. All these tools are mainly developed for reverse engineering C/C++ or Java programs. Some of them are language independent; they take a module dependency graph [26] or execution traces [24] as input. According to the reported case studies [26], however, these input models are also derived from programs that are developed with procedural or object oriented programming languages. In this work, we focus on reverse engineering PL/SQL programs. These programs are highly tangled with database operations and as such, program modules are subject to indirect data dependencies.

In our approach, we use a partitional algorithm, which produces flat decompositions. However, large software systems are usually decomposed according to a hierarchical structure. Software modules are grouped within packages each of which can be part of another package itself at a higher level. There exist hierarchical clustering algorithms [29] that can provide such a hierarchical decomposition. In this work, we did not utilize a hierarchical clustering algorithm. We aimed at discovering the top level decomposition only. However, these algorithms can also be used together with our approach in principle.

There exist only a few studies [30, 31] that focus on reverse engineering PL/SQL programs. One of these studies focus on deriving business rules from these programs [30]. In another study [31], data flow graphs are generated by analyzing PL/SQL source code and the accessed elements in the database. However, the derived graphs are used just for visualizing information and providing an abstract representation of the program. They are not processed further. In this work, we employ DSM [14] for representing the dependencies among PL/SQL programs. DSM is also used for providing a visual, abstract representation. However, our main goal is to apply clustering techniques on the derived DSM to infer a packaging structure for a PL/SQL program.

DSM has been applied for reasoning about software architectures and dependencies among the software modules [15, 17, 16]. However, it was either used for representing dependencies among high level modules/components of a system [15] or classes/packages of Java programs [17, 16]. Hereby, the type of dependencies that are documented are direct dependencies, e.g., the number of method calls from one class to another. In this work, we focus on PL/SQL programs. These programs also involve direct dependencies. Procedures and functions can make calls to each other. However, the actual coupling and cohesion among them can only be revealed based on their access patterns on database elements. To the best of our knowledge, DSM

structures that represent data dependencies have not been utilized for supporting reverse engineering and refactoring of PL/SQL programs.



## CHAPTER IV

### OVERALL APPROACH

In this thesis, we present two different approaches for clustering PL/SQL programs to reconstruct their software architecture design. The first one is named *Single DSM approach (SDSM)* and we refer to the second one as *Multi DSM approach (MDSM)*. In SDSM approach, only one DSM is used, which represents dependencies among the procedures and functions based on database tables that are commonly accessed. In MDSM approach, we used 3 different DSMs that represent other types of dependencies among the procedures and functions. First, we perform clustering according to each of these DSMs separately. Then, we perform cluster aggregation (meta clustering) on the obtained clustering results to propose a packaging structure to the designer. Both approaches are explained in detail in the following sections.

#### ***4.1 Single DSM Approach (SDSM)***

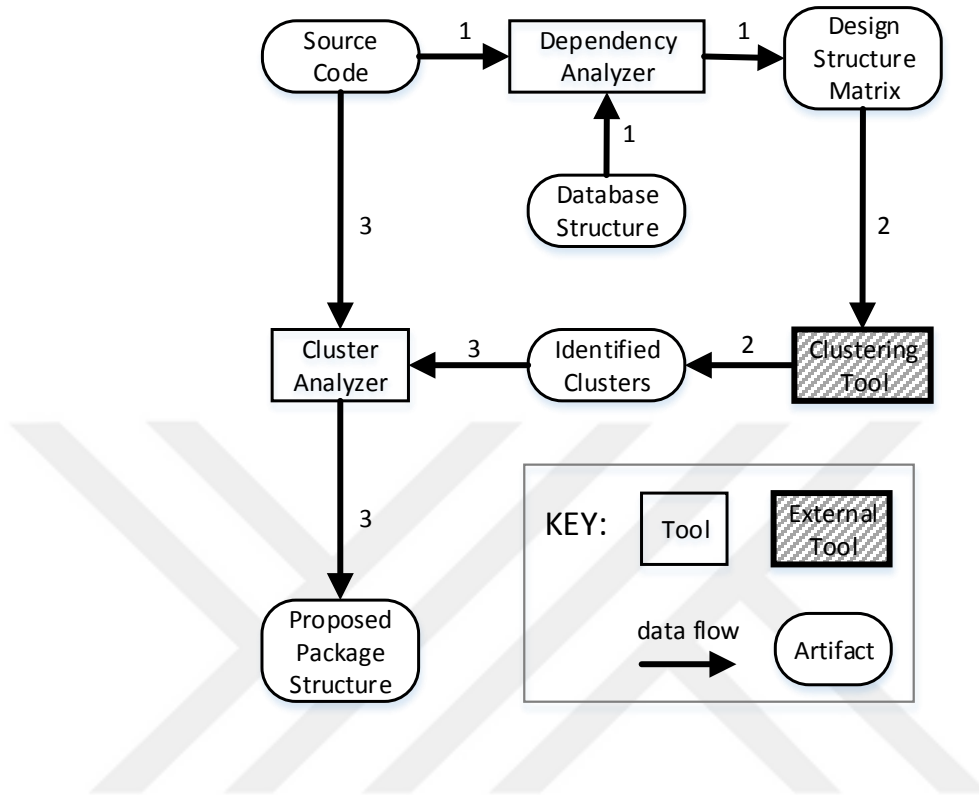
The overall SDSM approach, depicted in Figure 4, involves 3 steps. First, the program source code and the database structure (meta-data) is provided to our *Dependency Analyzer* tool as input (1). This tool creates a DSM that represents dependencies among the procedures and functions based on database tables that are commonly accessed. Second, the generated DSM is provided to an external tool for clustering (2). We employed the *Rapid Miner*<sup>1</sup> tool and support vector clustering [32] for this purpose. Finally, the identified clusters are processed by our tool *Cluster Analyzer* to propose a package structure for the analyzed source code (3).

The main steps executed by *Dependency Analyzer* is outlined in Algorithm 1.

---

<sup>1</sup><https://rapidminer.com/>





**Figure 4:** The overall SDSM approach.

Hereby, the terms procedure and function are used interchangeably as they are processed in the same way. There are in total  $N$  procedures (Line 1). Therefore, a DSM is initialized with  $N$  rows and  $N$  columns (Lines 2-4). Then, each pair of procedures are processed one by one (Lines 5-6). *Dependency Analyzer* finds the set of database tables that are accessed by these procedures (Lines 7-8). A procedure accesses a table if it performs any of the *insert*, *update*, *delete*, *select* operations on that table. Then the intersection of the two sets is obtained (Line 9), which is the set of commonly accessed tables for a pair. The size of this set is assigned as the dependency between that pair (Line 10). Note that we do not consider a direction for dependency. Hence, the generated DSM is symmetric in our case.

---

**Algorithm 1** Dependency analysis procedure.

---

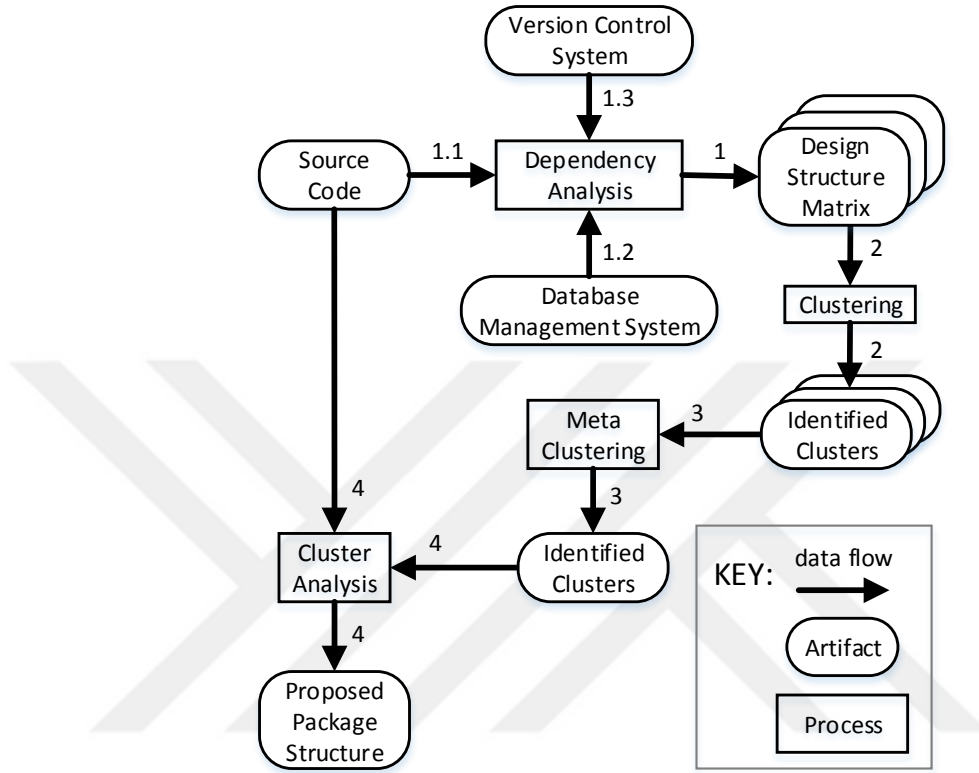
```
1:  $N \leftarrow$  the number of procedures
2: for  $i:=1$  to  $N$  do
3:    $DSM[i][i] \leftarrow 0$ 
4: end for
5: for  $n:=1$  to  $N-1$  do
6:   for  $k:=n+1$  to  $N$  do
7:      $T_n \leftarrow$  the set of tables accessed by procedure  $n$ 
8:      $T_k \leftarrow$  the set of tables accessed by procedure  $k$ 
9:      $I \leftarrow T_n \cap T_k$ 
10:     $DSM[n][k] \leftarrow DSM[k][n] \leftarrow |I|$ 
11:   end for
12: end for
```

---

## 4.2 Multi DSM Approach (MDSM)

The overall MDSM approach is depicted in Figure 5, which involves 4 steps. All of the involved processes are fully automated by tools. First, the program source code, the database management system and the version control system are provided to *Dependency Analysis* as input (1). The outcome of this process is a set of 3 DSM models. The first DSM model captures the set of database elements that are commonly accessed by system modules. The second DSM model captures model inter-dependency in terms of direct references in the source code. The third DSM model captures modifications to system modules that are applied by the same developer at the same time. In the second step, each of the generated DSM models are clustered (2). We employ the pivot algorithm, which was first proposed for solving the Clustering Aggregation [33] [18] problem. Similar algorithm is used in a previous study [34]. In the third step, we apply a second clustering phase in which the set of identified clusters are provided as input to meta-clustering (3). Finally, the identified clusters are processed to propose a package structure for the analyzed source code (4). In the following subsections, we explain dependency analysis, the employed clustering algorithm and our meta-clustering approach in detail. Then, in the next chapter, we explain the evaluation of both SDSM and MDSM approach in the context of two industrial case

studies.



**Figure 5:** The overall MDSM approach.

#### 4.2.1 Dependency Analysis

Here, we explain the dependency analysis process. The main steps executed by *Dependency Analyzer* is outlined in Algorithm 2. Hereby, the terms procedure and function are used interchangeably as they are processed in the same way. In this step we dependency analyzer creates 3 different DSM. There are in total  $N$  procedures (Line 1). Therefore, a DSM1, DSM2 and DSM3 are initialized with  $N$  rows and  $N$  columns (Lines 2-6). Then, each pair of procedures is processed one by one (Lines 7-8).

For DSM1 *Dependency Analyzer* finds the set of database tables that are accessed by these procedures (Lines 9-10). A procedure accesses a table if it performs any of

---

**Algorithm 2** Dependency analysis procedure.

---

```
1:  $N \leftarrow$  the number of procedures
2: for  $i:=1$  to  $N$  do
3:    $DSM1[i][i] \leftarrow 0$ 
4:    $DSM2[i][i] \leftarrow 0$ 
5:    $DSM3[i][i] \leftarrow 0$ 
6: end for
7: for  $n:=1$  to  $N-1$  do
8:   for  $k:=n+1$  to  $N$  do
9:      $T_n \leftarrow$  the set of tables accessed by procedure  $n$ 
10:     $T_k \leftarrow$  the set of tables accessed by procedure  $k$ 
11:     $I \leftarrow T_n \cap T_k$ 
12:     $DSM1[n][k] \leftarrow DSM1[k][n] \leftarrow |I|$ 
13:
14:     $O_n \leftarrow$  the set of objects accessed by procedure  $n$ 
15:     $O_k \leftarrow$  the set of objects accessed by procedure  $k$ 
16:     $I \leftarrow O_n \cap O_k$ 
17:     $DSM2[n][k] \leftarrow DSM2[k][n] \leftarrow |I|$ 
18:
19:     $S_n \leftarrow$  the set of modifications made on procedure  $n$ , where each modification
    is represented by a tuple  $M = \langle \text{date (dd.mm.yyyy format)}, \text{developer} \rangle$ 
20:     $S_k \leftarrow$  the set of modifications made on procedure  $k$ , where each modification
    is represented by a tuple  $M = \langle \text{date (dd.mm.yyyy format)}, \text{developer} \rangle$ 
21:     $I \leftarrow S_n \cap S_k$ 
22:     $DSM3[n][k] \leftarrow DSM3[k][n] \leftarrow |I|$ 
23:   end for
24: end for
```

---

the *insert, update, delete, select* operations on that table. Then the intersection of the two sets is obtained (Line 11), which is the set of commonly accessed tables for a pair. The size of this set is assigned as the dependency between that pair for DSM1 (Line 12).

For DSM2 *Dependency Analyzer* finds the set of objects that are accessed by these procedures (Lines 14-15). A procedure accesses a object if it performs any operations on that object. Then the intersection of the two sets is obtained (Line 16), which is the set of commonly accessed objects for a pair. The size of this set is assigned as the dependency between that pair for DSM2 (Line 17).

For DSM3 *Dependency Analyzer* finds the set of modifications to procedures that are applied by the same developer at the same day (Lines 19-20). Then the intersection of the two sets is obtained (Line 21) . The size of this set is assigned as the dependency between that pair for DSM3 (Line 22).

Note that we do not consider a direction for dependency. Hence, the all generated DSMs are symmetric in our case.

#### 4.2.2 Pivot Clustering

In this section we explain the clustering approach. We used the pivot algorithm for clustering procedures according to each DSM. The main steps executed by Pivot Clustering is outlined in Algorithm 3. The DSM and a threshold is given as input to the pivot function (Line 1) There are in total N procedures (Line 2). Procedure list P is initialized (Line 3-5). P is re-created after randomly shuffling the initial P (Line 6).

<sup>2</sup> Each element of P is processed while size of P is greater than 0 (Line 7). A cluster created with member of first processed procedure (Line 8). Then, each procedures

---

<sup>2</sup>There are other approaches for choosing the pivot member randomly such as in [34]. In their study Gürsun et al. use a randomized algorithm since at every recursive call it picks a random prefix to play the role of a pivot. However we prefer initial shuffling in order to comply with our DSM structure since this approach also introduces randomization into the pivot selection procedure as well.

with processed procedure we check DSM cell value between them compared the cell value with the given threshold ( $\tau$ ) (Line 9-10). If the cell value is greater than the threshold, that procedure is added to cluster C (Line 11). After completing the set, P is re-created with P minus C (Line 14), Then all stuff is repeated until no member remains (Lines 7-15).

In our cases Pivot Clustering is used for each 3 DSMs and we get 3 clustering results. Threshold value is given 2 for first and third DSMs and 1 for second DSM. Because second DSM is a bit sparse. Then we used these clusters for cluster aggregation (meta-clustering) to obtain single cluster which explained in section 4.2.3.

---

**Algorithm 3** Pivot Clustering.

---

```

1: Function{Pivot}{ $DSM[][]$ ,  $\tau$  }
2:  $N \leftarrow$  the number of procedures
3: for  $i:=1$  to  $N$  do
4:    $P[i] \leftarrow [i]$ 
5: end for
6:  $P \leftarrow$  randomly shuffle  $P$ 
7: while ( $|P| > 0$ ) do
8:   create a cluster  $C$  with member of  $P[1]$ 
9:   for  $n:=2$  to  $|P|$  do
10:    if  $DSM[P[1]][n] \geq \tau$  then
11:      Add to  $n$  to Cluster  $C$ 
12:    end if
13:  end for
14:   $P = P \setminus C$ 
15: end while
16: End Function

```

---

### 4.2.3 Meta Clustering

In this section we explain the meta clustering approach. The main steps executed by Meta Clustering is outlined in Algorithm 4. There are in total N procedures (Line 1). The count variable is initialized with 0 (Line 2). There are in total clusters (Line 3). Then, each pair of procedures is processed one by one (Lines 4-5) . Then, for each cluster (Line 6), if each pair of procedures are in same cluster then count is

incremented (Line 7-8). Then new similarity DSM is generated with this count value (Line 11). The count variable is reset for next iteration (Line 12). After similarity DSM completed, finally this DSM is provided to Pivot algorithm as input to generate final single clustering, here threshold value is given as half of sizeOfCluster which is 3 in our case (Line 13).

---

**Algorithm 4** Meta Clustering.

---

```

1:  $N \leftarrow$  the number of procedures
2:  $count \leftarrow 0$ 
3:  $C[] \leftarrow Clusters$ 
4: for  $n:=1$  to  $N-1$  do
5:   for  $k:=n+1$  to  $N$  do
6:     for  $i:=1$  to  $|C|$  do
7:       if  $n$  and  $k$  are in same cluster then
8:          $count \leftarrow count + 1$ 
9:       end if
10:    end for
11:     $DSM[n][k] \leftarrow DSM[k][n] \leftarrow count$ 
12:     $count \leftarrow 0$ 
13:  end for
14: end for
15: Pivot(DSM,  $\frac{|C|}{2}$ )

```

---

In the next chapter we introduce industrial case studies and illustrate all the steps of our approach in the context of these case studies. We also explain the employed tools, models and techniques in more detail.

## CHAPTER V

### INDUSTRIAL CASE STUDY

In this section we present industrial case studies for automatically clustering modules of a legacy application implemented with the PL/SQL language. We have applied approaches on two large-scale PL/SQL programs developed and being maintained by Turkcell<sup>1</sup>, which is the largest GSM operator in Turkey. The analyzed applications are a Customer Relation Management (CRM) system and a Billing system. We have applied our two approaches, which are SDSM and MDSM to each systems separately. Firstly, we illustrate the application of our approaches for these systems. Then, we will discuss the results. We can not share real procedure and package names due to confidentiality; however, we will present abstracted artifacts and results.

#### *5.1 Legacy Systems*

In this section we explain two important legacy systems for Turkcell which are being developed by the Turkcell development team. Each system is operational since 1993.

##### **5.1.1 CRM System**

CRM system is our first subject system. Approximately half of its source code is developed with the PL/SQL language. The code base of this system is maintained by Turkcell. The system comprises more than 1.800 KLOC PL/SQL code in total. It is operational since 1993, serving more than 10000 users.

---

<sup>1</sup><http://www.turkcell.com.tr>



### 5.1.2 Billing System

Our second subject system is Turkcell Billing system. Approximately half of the source code of this system is developed with the PL/SQL language as well. The code base of this system is also maintained by Turkcell. The system comprises more than 1.950 KLOC PL/SQL code in total. It is operational since 1993, serving more than 16 million post paid subscribers. Every month post paid customer invoices are being prepared via this system.

## 5.2 Case Studies

As explained in Chapter 4 we have two different approaches. We applied two approaches to each legacy system which explained above separately and we get promising results.

### 5.2.1 SDSM Approach on CRM System

In this case study, we analyzed one of the main schemas of the CRM system, which consists of 157 stored procedures and 659 tables. Therefore, the generated DSM included 157 rows and 157 columns. The DSM model captures the set of database elements that are commonly accessed by system modules. Then, we applied support vector clustering on this DSM by using the parameters listed in Table 2.

The first cropped snapshot of the clustered DSM is depicted in Figure 6, which shows the first 33 rows and 33 columns of the matrix. We can also see an identified cluster in this part of the matrix between rows/columns 24 and 30.

### 5.2.2 SDSM Approach on Billing System

In our this case study, we analyzed one of the main schemas of the Billing system, which consists of 150 stored procedures and 1194 tables. Therefore, the generated DSM included 150 rows and 150 columns. The DSM model captures the set of

**Table 2:** Support Vector Clustering parameters

parameter	value
<i>min pts</i>	<i>2</i>
<i>kernel type</i>	<i>radial</i>
<i>kernel gamma</i>	<i>1.0</i>
<i>kernel cache</i>	<i>200</i>
<i>convergence epsilon</i>	<i>0.001</i>
<i>max iteration</i>	<i>100000</i>
<i>p</i>	<i>0.0</i>
<i>r</i>	<i>-1.0</i>
<i>number sample points</i>	<i>20</i>

database elements that are commonly accessed by system modules. Then, we applied support vector clustering on this DSM by using the parameters listed in Table 2.

The cropped snapshot of the clustered DSM is depicted in Figure 9, which shows the first 36 rows and 36 columns of the matrix. We can also see an identified cluster in this part of the matrix between rows/columns 8 and 12.

### 5.2.3 MDSM Approach on CRM System

In our this case study, we analyzed same schema of the CRM system mentioned in 5.2.1. But here our tool generated 3 different DSMs for the system. The first DSM model is the same DSM that is used first case study in 5.2.1 which captures the set of database elements that are commonly accessed by system modules. The second DSM model created captures model inter-dependency in terms of direct references in the source code. The third DSM model captures modifications to system modules that are applied by the same developer at the same time. After generation these DSMs, we perform pivot clustering according to each of these matrices separately. Then, we perform cluster aggregation (meta clustering) on the obtained clustering results to propose a packaging structure to the designer.

The cropped snapshot of the clustered second and third DSMs are depicted in Figure 7 and in Figure 8, which shows the partial rows and columns of the matrix

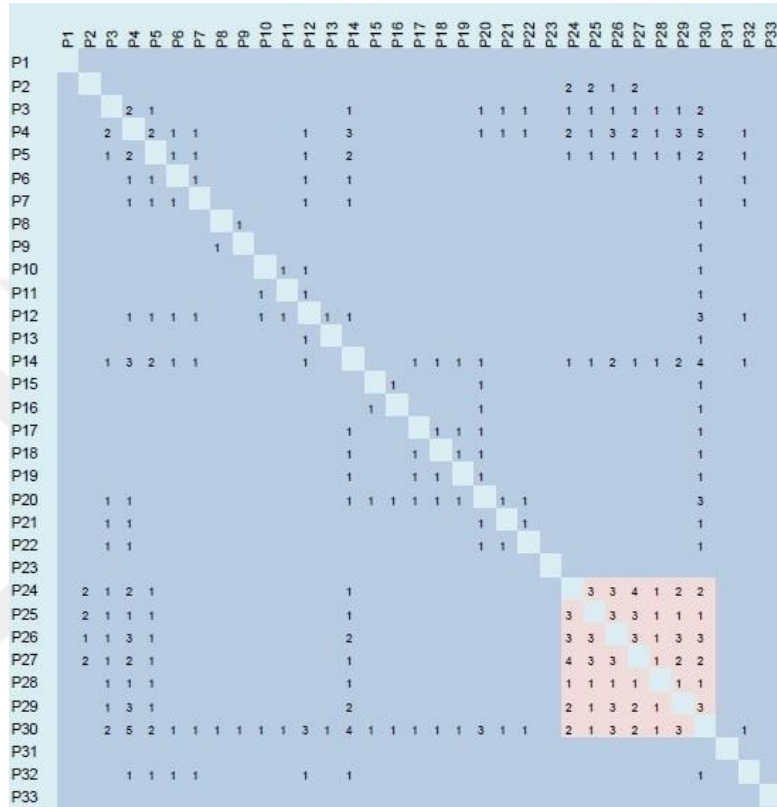
are shown in Appendices A section. As shown there second DSM in Figure 7 is very sparse matrix. This show us there is not big inter-dependency in terms of direct references in the source code for CRM system.

#### 5.2.4 MDSM Approach on Billing System

In our this case study, we analyzed same schema of the Billing system mentioned in 5.2.2. Our tool generated 3 different DSMs for this system as mentioned in 5.2.3. Each of DSMs captures a different type of dependency among the Billing system modules. After generation these DSMs, we perform pivot clustering according to each of these matrices separately. Then, we perform cluster aggregation (meta clustering) on the obtained clustering results to propose a packaging structure to the designer.

The cropped snapshot of the clustered second and third DSMs are depicted in Figure 10 and in Figure 11, which shows the partial rows and columns of the matrix are shown in Appendices B section. As shown there second DSM in Figure 10 is very sparse matrix. This show us there is not big inter-dependency in terms of direct references in the source code for Billing system too.

For all case studies, these structures are further analyzed by *Cluster Analyzer* to map it back to the source code and derive a package structure. We present and discuss the results in the following section.



**Figure 6:** A snippet from the generated DSM model, capturing the set of database elements that are commonly accessed by system modules for CRM system.

# CHAPTER VI

## RESULTS AND DISCUSSION

### 6.1 Results

For our first case study which we applied SDSM approach using support vector clustering via Rapid Miner for CRM system, in total 7 clusters were derived as listed in Table 3. Hereby, the number of items represent the number of procedures and functions that are placed in the same cluster. For instance, *Cluster 5* includes 8 procedures. These procedures were not belonging to any package in the original application. They were defined as standalone procedures although they were working on the same database tables. We have validated this result with 4 different domain experts, all of whom agreed that these procedures perform related tasks and they should have been placed in the same package. The results regarding the clusters 2, 3, 4 and 6 were also validated likewise.

**Table 3:** *SDSM Cluster Analyzer* results for CRM System

Cluster	Number of Items
Cluster 0	77
Cluster 1	48
Cluster 2	3
Cluster 3	3
Cluster 4	12
Cluster 5	8
Cluster 6	6
Total	157

One can notice that clusters 0 and 1 include many items. The tool was basically unable to differentiate these items further. They are neither related nor distinguished in terms of the database tables they access.

For our other case study which we applied again SDSM approach using support

vector clustering via Rapid Miner for Billing system, in total 6 clusters were derived as listed in Table 4. Hereby, the number of items also represent the number of procedures and functions that are placed in the same cluster. For instance, *Cluster 2* includes 8 procedures. These procedures were not belonging to any package in the original application. They were defined as standalone procedures although they were working on the same database tables. We have validated this result with 4 different domain experts, all of whom agreed that these procedures perform related tasks and they should have been placed in the same package. The results regarding the clusters 3, 4 and 5 were also validated likewise.

**Table 4:** *SDSM Cluster Analyzer* results for Billing System

Cluster	Number of Items
Cluster 0	75
Cluster 1	52
Cluster 2	8
Cluster 3	8
Cluster 4	3
Cluster 5	4
Total	150

In this case clusters 0 has 75 items. The tool was basically unable to differentiate some items further here too.

For our other case study which we applied MDSM approach using pivot clustering via Cluster Aggregation for CRM system, in total 17 clusters were derived as listed in Table 5. We can say that we get better results than compared the first case study. Because while only 32 procedures are clustered with the first approach, now in total 87 procedures are clustered. The tool was now differentiated these items further which are not clustered in the first case. We have also validated this result again with 4 different domain experts, all of whom agreed that these procedures perform related tasks and they should have been placed in the same package.

Finally, for our last case study which we applied MDSM approach using pivot

**Table 5:** *MDSM Cluster Analyzer* results for CRM System

Cluster	Number of Items
Cluster 0	42
Cluster 1	2
Cluster 1	2
Cluster 2	2
Cluster 3	2
Cluster 4	2
Cluster 5	2
Cluster 6	2
Cluster 7	2
Cluster 8	2
Cluster 9	2
Cluster 10	2
Cluster 11	3
Cluster 12	3
Cluster 13	5
Cluster 14	5
Cluster 15	9
Cluster 16	70
Total	157

clustering via Cluster Aggregation for Billing system, in total 18 clusters were derived as listed in Table 6.

## 6.2 *Discussions*

As mentioned above we have two approaches, SDSM and MDSM. We applied both approaches to legacy systems separately and we get promising results. In the first approach (SDSM), the tool couldn't differentiate some items further as mentioned above. This problem was the main motivation for developing MDSM. By means of MDSM, our tool differentiated these items and we got better results. We compared the results obtained by using MDSM with the results obtained using SDSM. MDSM approach succeeded on average 30% better in terms of the percentage of procedures that are confirmed to be clustered correctly in a package. But we observed that even using the MDSM approach the tool couldn't still differentiate some items further as

**Table 6:** *MDSM Cluster Analyzer* results for Billing System

Cluster	Number of Items
Cluster 0	95
Cluster 1	2
Cluster 2	2
Cluster 3	2
Cluster 4	2
Cluster 5	2
Cluster 6	2
Cluster 7	2
Cluster 8	2
Cluster 9	2
Cluster 10	2
Cluster 11	3
Cluster 12	3
Cluster 13	4
Cluster 14	4
Cluster 15	4
Cluster 16	6
Cluster 17	11
Total	150

well. Each approach can be used standalone separately and additionally they can be used serially, because they are complements of each other. So we suggest that each approach should be used together. First SDSM approach can be used and then MDSM tool should be used if necessary.

### ***6.3 Threats to Validity***

There are some validity threats to our evaluation. First, It is based on subjective expert opinion rather than quantitative measurements. We tried to mitigate this threat by consulting 4 different domain experts who are software architects and they are very skilled people in their area and have at least 10 years of professional experience. It is essential to have at least 10 years of experience in a specific domain and being actively involved in at least 10 big projects to become a domain expert at Turkcell. A second threat is regarding the use of only one main schema for each of the two systems



for the industrial case study and these schemas are not very big size. Therefore, we plan to perform more case studies which are bigger size in the future.

In this work, we focused on PL/SQL programs. In fact, we plan to integrate our approach in standard tools that are used for developing and maintaining these programs. However, our approach is relevant and applicable for any type of program that is highly coupled with a database management system.



## CHAPTER VII

### CONCLUSIONS AND FUTURE WORK

In this thesis, we provide two approaches and toolset that supports automated modularization of software systems by considering different types of dependencies.

In our first approach, we derive (indirect) dependencies among PL/SQL procedures by analyzing the accessed database elements. We represent these dependencies in the form of a design structure matrix. Then, we cluster the procedures and propose a packaging structure to the designer. An initial evaluation of this approach showed promising results in which most of the procedures were successfully clustered in relevant packages. An analysis of the remaining procedures revealed that some of them directly refer to each other or they use common resources other than database elements. Therefore, we introduced a second approach that utilizes input regarding multiple types of dependencies at the same time. We construct multiple dependency matrices each of which captures a different type of dependency among the system modules. First, we perform clustering according to each of these matrices separately. Then, we perform cluster aggregation (meta clustering) on the obtained clustering results to propose a packaging structure to the designer. Procedures are clustered according to these dependencies to propose a package structure for grouping procedures.

We performed two industrial case studies from the telecommunication domain. We observed promising results, in which several package suggestions were confirmed to be necessary by domain experts. We conclude that our approaches and toolset can support the re-factoring of legacy applications for better modularity and maintainability.

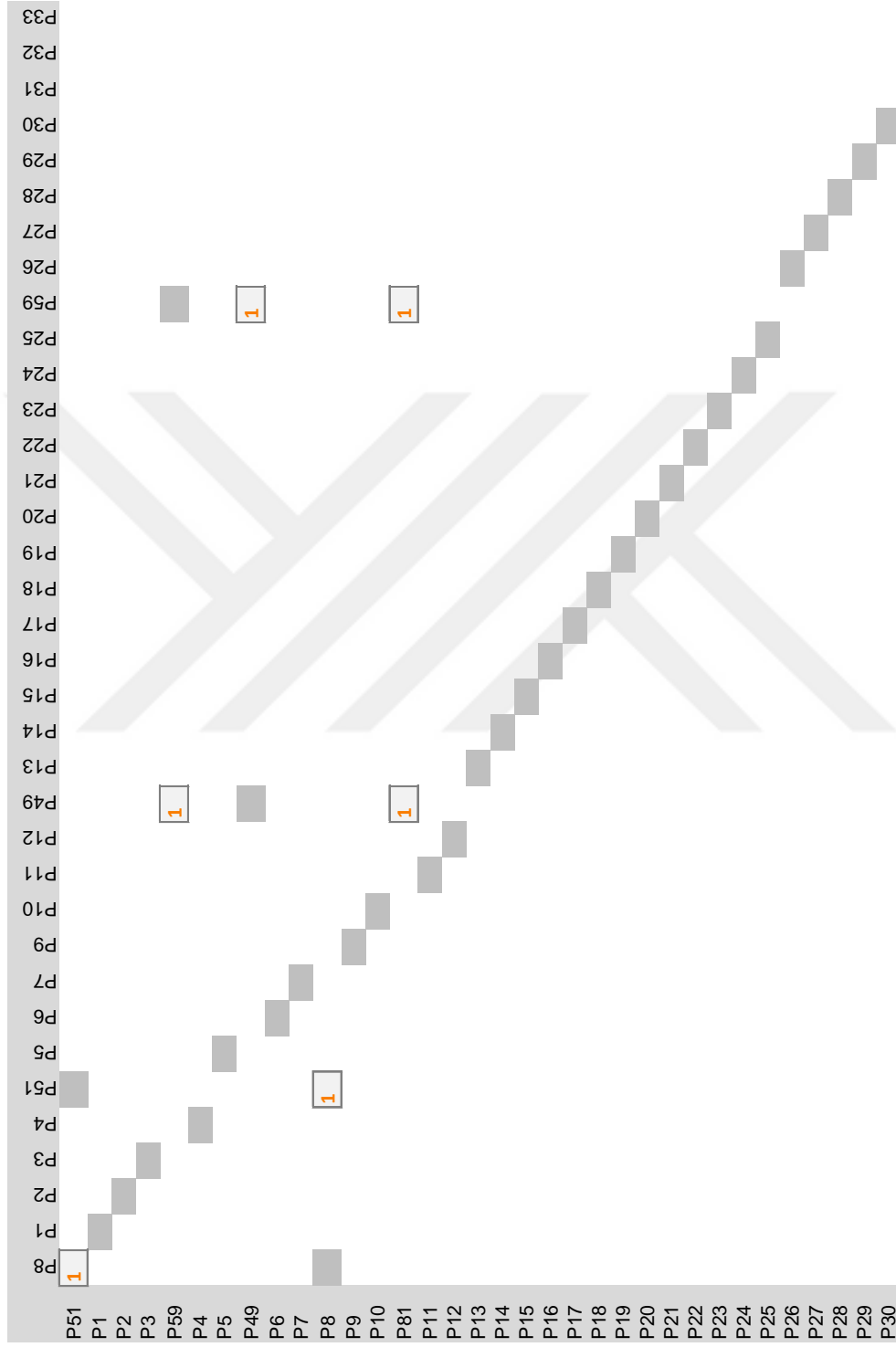
In our future work, we plan to extend our toolset to new approaches to increase the accuracy. We also plan to conduct more case studies.



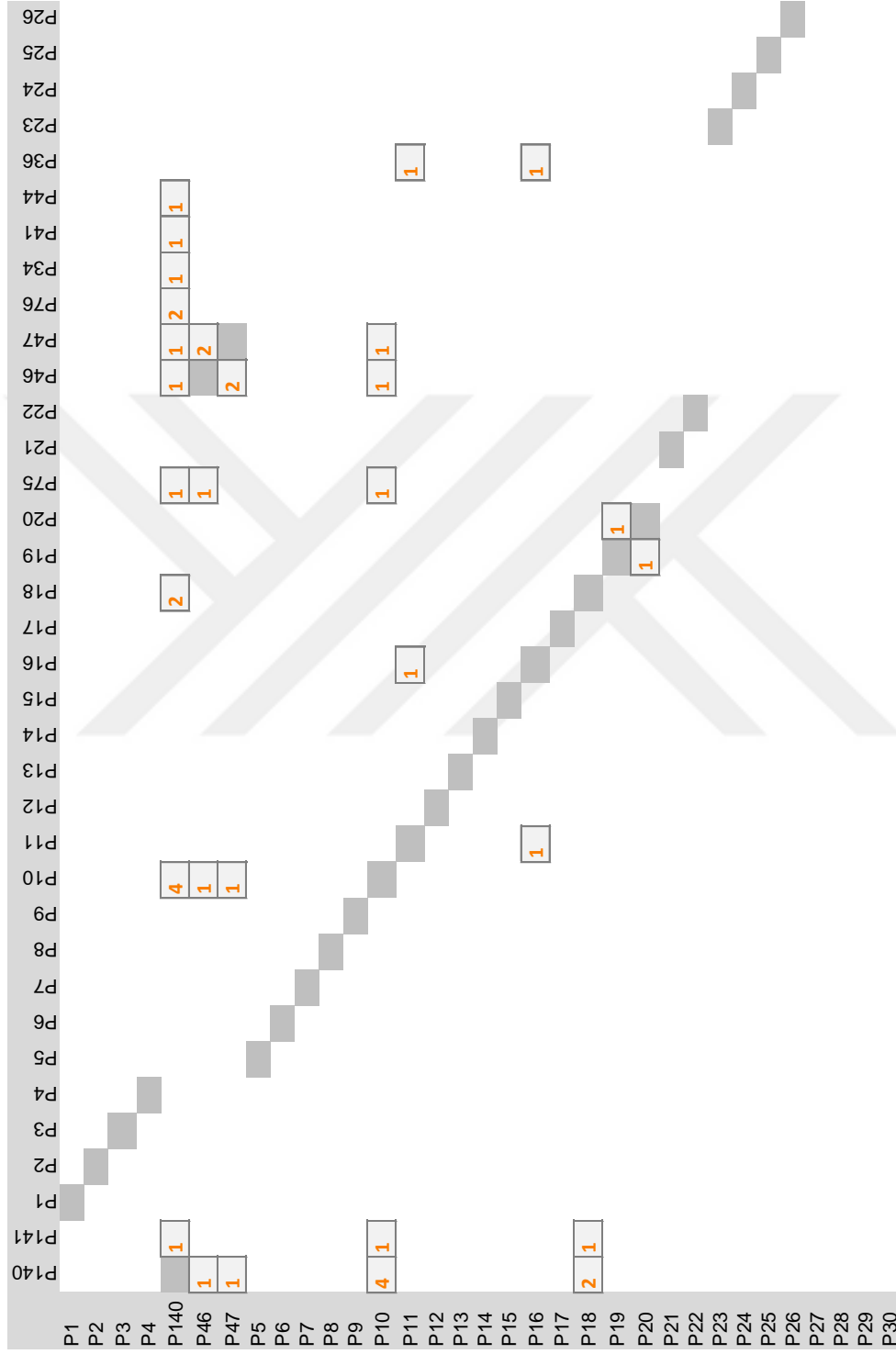
# APPENDIX A

## GENERATED DSM MODELS FOR CRM





**Figure 7:** A snippet from the generated DSM model, capturing model inter-dependency in terms of direct references in the source code for the CRM system. (The rows and columns are reordered to highlight cells that have the value 1 in the sparse matrix)

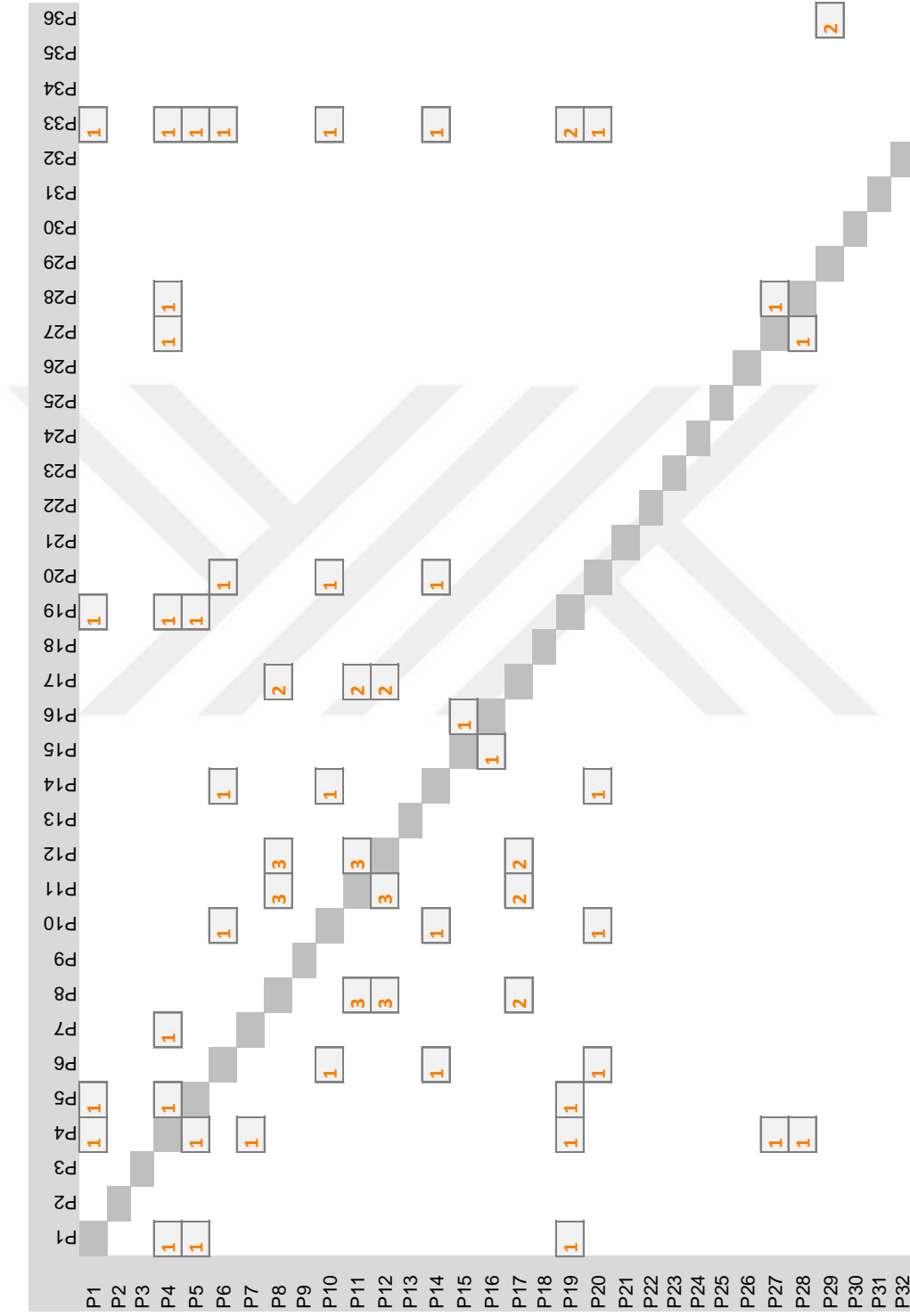


**Figure 8:** A snippet from the generated DSM model, capturing modifications to system modules that are applied by the same developer at the same time for the CRM system. (The rows and columns are reordered to highlight cells that have the value greater than 0 in the matrix)

## APPENDIX B

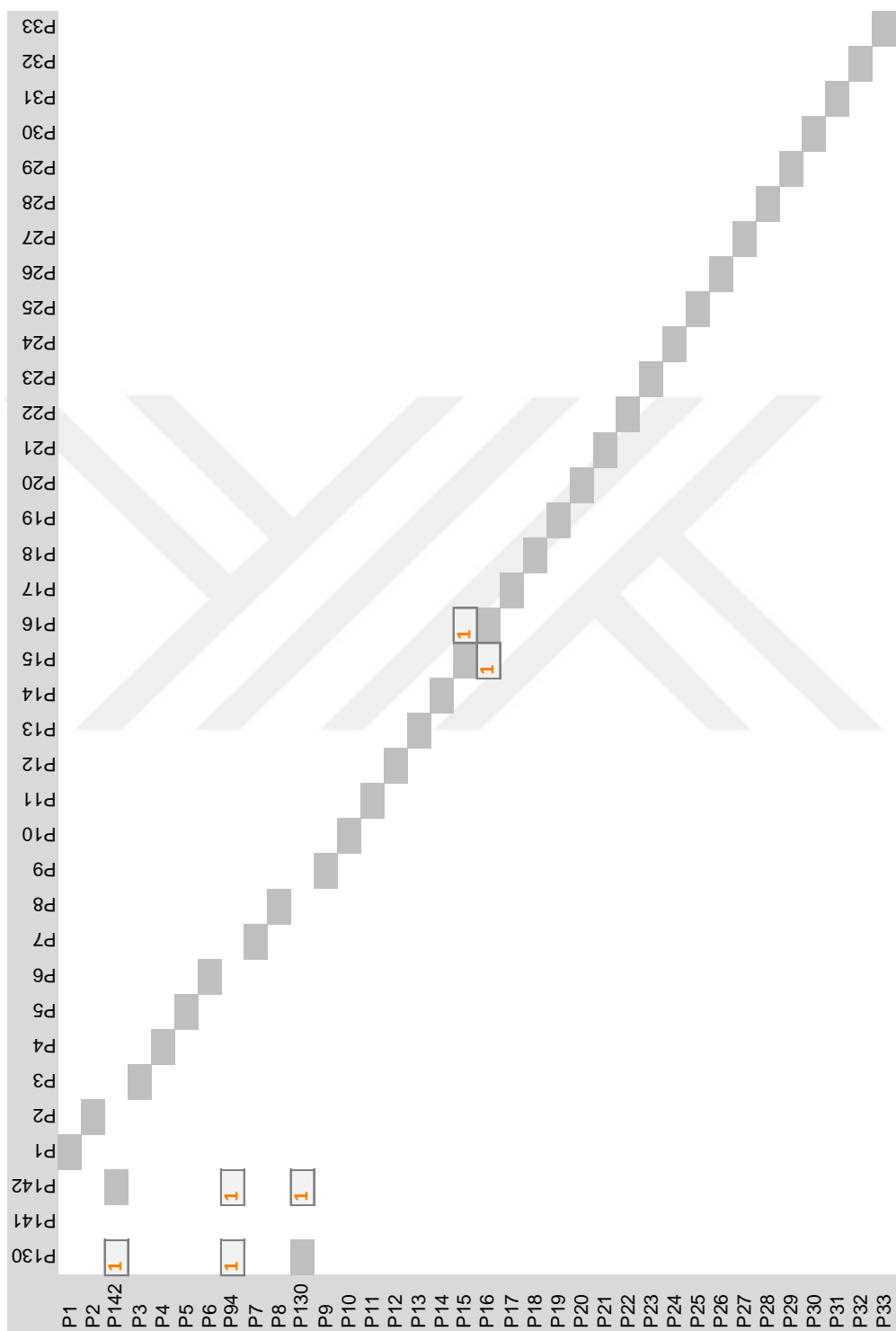
### GENERATED DSM MODELS FOR BILLING



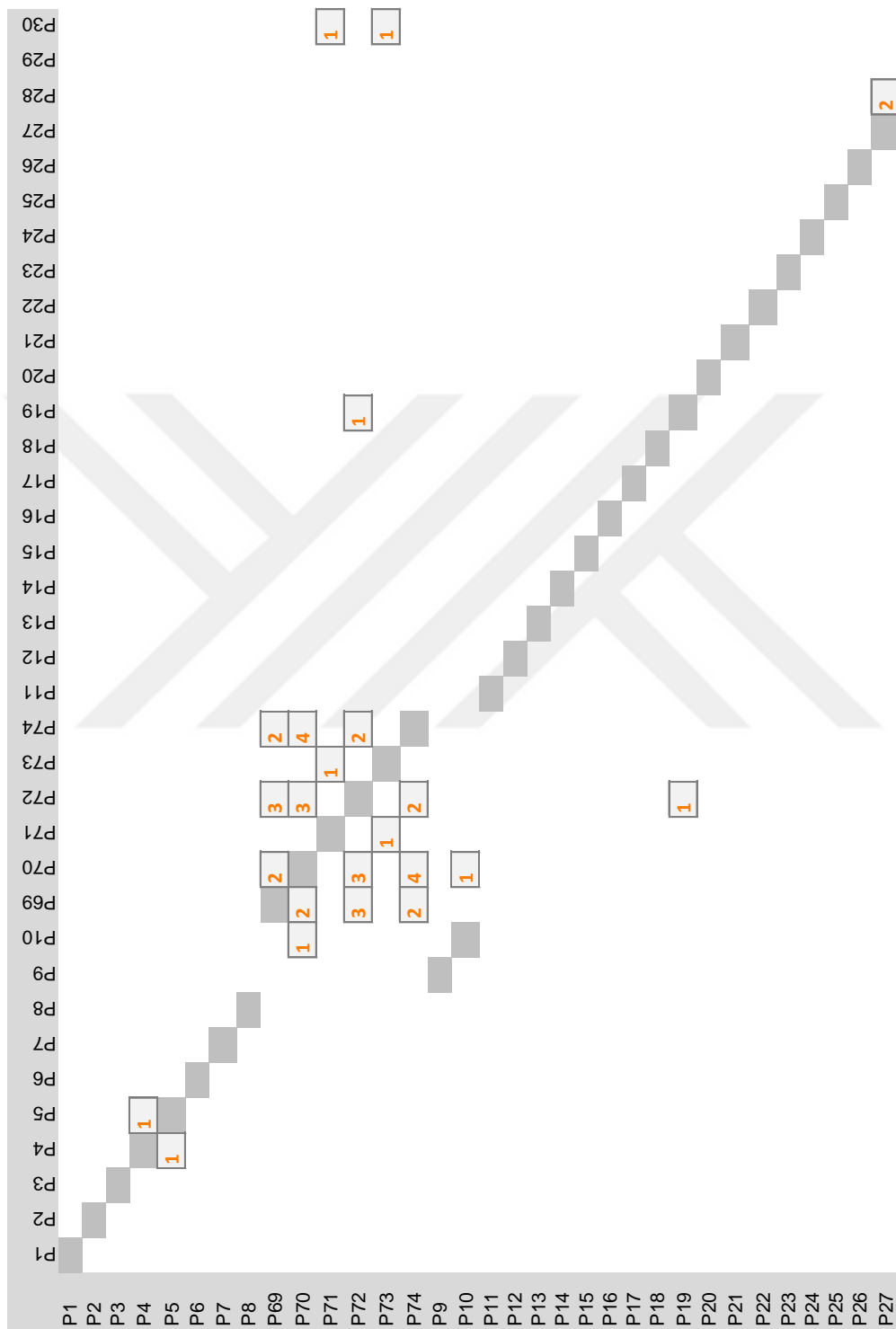


**Figure 9:** A snippet from the generated DSM model, capturing the set of database elements that are commonly accessed by system modules for the Billing system.





**Figure 10:** A snippet from the generated DSM model, capturing model inter-dependency in terms of direct references in the source code for the Billing system. (The rows and columns are reordered to highlight cells that have the value 1 in the sparse matrix)



**Figure 11:** A snippet from the generated DSM model, capturing modifications to system modules that are applied by the same developer at the same time for the Billing system. (The rows and columns are reordered to highlight cells that have the value greater than 0 in the matrix)

## References

- [1] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
- [2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 3 ed., 2003.
- [3] R. Taylor, N. Medvidovic, and E. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. 2009.
- [4] P. Clements et al., *Documenting Software Architectures*. Addison-Wesley, 2002.
- [5] S. Eick et al., “Does code decay? assessing the evidence from change management data,” *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1 – 12, 2001.
- [6] G. Murphy, D. Notkin, and K. Sullivan, “Software reflexion models: Bridging the gap between design and implementation,” *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364 – 308, 2001.
- [7] M. Nelson, “A survey of reverse engineering and program comprehension,” *CoRR*, vol. abs/cs/0503068, 2005.
- [8] S. Ducasse and D. Pollet, “Software architecture reconstruction: A process-oriented taxonomy,” *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573 – 591, 2009.
- [9] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, “Automatic package coupling and cycle minimization,” in *Proceedings of the 16th Working Conference on Reverse Engineering*, pp. 103–112, 2009.
- [10] K. Praditwong, M. Harman, and X. Yao, “Software module clustering as a multi-objective search problem,” *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.
- [11] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, “Using structural and semantic measures to improve software modularization,” *Empirical Software Engineering*, vol. 18, no. 5, pp. 901 – 932, 2013.
- [12] C. Patel, A. Hamou-Lhadj, and J. Rilling, “Software clustering using dynamic analysis and static dependencies,” in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pp. 27–36, 2009.
- [13] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, “Software dependencies, work dependencies, and their impact on failures,” *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, pp. 864–878, 2009.

- [14] S. Eppinger and T. Browning, *Design Structure Matrix Methods and Applications*. Cambridge, MA, USA: MIT Press, 2012.
- [15] K. Sullivan, Y. Cai, B. Hallen, and W. Griswold, “The structure and value of modularity in software design,” in *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 99–108, 2001.
- [16] R. Sangwan and C. Neill, “Characterizing essential and incidental complexity in software architectures,” in *Proceedings of the Joint Working IEEE/IFIP Conference on European Conference on Software Architecture*, pp. 265–268, 2009.
- [17] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, “Using dependency models to manage complex software architecture,” in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 167–176, 2005.
- [18] A. Gionis, H. Mannila, and P. Tsaparas, “Clustering aggregation,” *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 4, 2007.
- [19] S. Vega-Pons and J. Ruiz-Shulcloper, “A survey of clustering ensemble algorithms,” *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 25, no. 03, pp. 337–372, 2011.
- [20] B. Abu-Jamous, R. Fa, D. J. Roberts, and A. K. Nandi, “Paradigm of tunable clustering using binarization of consensus partition matrices (bi-copam) for gene discovery,” *PLoS One*, vol. 8, no. 2, p. e56432, 2013.
- [21] “Oracle Database Online Documentation 11g Release developing and using stored procedures.” [http://docs.oracle.com/cd/B28359\\_01/appdev.111/b28843/tdddg\\_procedures.htm](http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm). Accessed in September 2015.
- [22] R. Koschke, “Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 2, pp. 87–109, 2003.
- [23] G. Guo, J. Atlee, and R. Kazman, “A software architecture reconstruction method,” in *Proceedings of the First Working Conference on Software Architecture*, (Deventer, The Netherlands, The Netherlands), pp. 15–34, 1999.
- [24] T. Callo, P. America, and P. Avgeriou, “A top-down approach to construct execution views of a large software-intensive system,” *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 233–260, 2013.
- [25] K. O. J. Ferrante and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

- [26] B. Mitchell and S. Mancoridis, “On the automatic modularization of software systems using the bunch tool,” *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193 – 208, 2006.
- [27] L. Qingshan et al., “Architecture recovery and abstraction from the perspective of processes,” in *WCRE*, pp. 57–66, 2005.
- [28] C. Sun, J. Zhou, J. Cao, M. Jin, C. Liu, and Y. Shen, “ReArchJBs: a tool for automated software architecture recovery of javabeans-based applications,” in *Proceedings of the 16th Australian Software Engineering Conference*, pp. 270–280, 2005.
- [29] O. Maqbool and H. Babri, “Hierarchical clustering for software architecture recovery,” *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.
- [30] O. Chaparro, J. Aponte, F. Ortega, and A. Marcus, “Towards the automatic extraction of structural business rules from legacy databases,” in *Proceedings of the 19th Working Conference on Reverse Engineering*, pp. 479–488, 2012.
- [31] M. Habringer, M. Moser, and J. Pichler, “Reverse engineering PL/SQL legacy code: An experience report,” in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pp. 553–556, 2014.
- [32] A. Ben-Hur, D. Horn, H. Siegelmann, and V. Vapnik, “Support vector clustering,” *The Journal of Machine Learning Research*, vol. 2, no. 12, pp. 125–137, 2002.
- [33] N. Ailon, M. Charikar, and A. Newman, “Aggregating inconsistent information: ranking and clustering,” *Journal of the ACM (JACM)*, vol. 55, no. 5, p. 23, 2008.
- [34] G. Gürsun, N. Ruchansky, E. Terzi, and M. Crovella, “Routing state distance: a path-based metric for network analysis,” in *IMC*, 2012.