Tool Support and Topological Study of Schema Evolution in terms of Foreign Keys

A Thesis

submitted to the designated

by the General Assembly of Special Composition

of the Department of Computer Science and Engineering

Examination Committee

by

Konstantinos Dimolikas

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION

IN SOFTWARE

University of Ioannina

January 2019

Examining Committee:

- **Panos Vassiliadis**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina (Supervisor)
- **Evaggelia Pitoura**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Apostolos Zarras**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina

DEDICATION

To my family.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisor Panos Vassiliadis for his guidance and the fruitful collaboration we had throughout my graduate studies in the University of Ioannina. Finally, I should express my gratitude to my parents for all the support and encouragement they offered me all these years.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Konstantinos Dimolikas. MSc in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, January 2019.

Tool Support and Topological Study of Schema Evolution in terms of Foreign Keys

Advisor: Panos Vassiliadis, Associate Professor.

Studying the evolution of databases' structure, known as *schema evolution*, is of great importance, since it can reveal patterns that will help administrators devote less time for increasing databases' information capacity with the least possible effects on the surrounding applications and take all the necessary maintenance actions for preserving and enhancing databases' performance.

The main research question that we attempt to answer in this Thesis can be expressed in this way: *is there a relationship between tables' involvement with foreign keys and their evolution?* For answering this question, we adopt a model that considers each version of a schema as a graph which includes schema's tables and foreign key constraints as nodes and edges, respectively. The union of the graphs forms the *Diachronic Graph*, which comprises all the tables and all the foreign keys that ever exist in schema's history. We also define four categories, namely *isolated*, *source*, *lookup* and *internal*, for the tables with respect to the combination of their in- and out- degrees in the Diachronic Graph. We refer to these categories with the term *topological* since they describe the arrangement of the tables in the Diachronic Graph with respect to their inciting foreign keys. We then classify tables into the topological categories and we study how tables' topology is associated with several evolution-related properties, such as tables' duration, their update activity, their size change, etc. The schema histories that we utilize in the context of our work derive from 5 relational databases supporting open-source projects.

The most significant results of our research work, which are also verified by the statistical evidence, are: (a) a relationship between tables' topological categories and their probability to be born in the originating version of their databases and (b) a correlation between tables' topology and their update activity. Specifically, we have identified that the more topologically complex a table is the more intense is its life in terms of its update activity and the higher is the probability to be introduced in the very first version of its schema history.

To facilitate the research part of the Thesis, we perform an extensive refactoring in the architecture of the Parmenidian Truth tool that visualizes the schema evolution of relational databases. After identifying and prioritizing design defects, we have applied a series of modifications in the source code of the tool, aiming at increasing tool's extendibility potential. To verify that the changes we introduced have not altered tool's expected behavior, we have created unit tests for all the modules we either modified or added. Finally, we have evaluated the enhancements of the refactoring process by comparing the design quality of the tool before and after the refactoring.

Complementing the refactoring of the tool, we have also constructed a web application that visualizes the schema evolution of relational databases and summarizes the main corresponding statistics.

Εκτεταμενη Περιληψη στα Ελληνικα

Κωνσταντίνος Δημολίκας, ΜΔΕ στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιανουάριος 2019.

Μελέτη της εξέλιξης σχήματος βάσεων δεδομένων σε σχέση με τα ξένα κλειδιά με τη χρήση εξειδικευμένου λογισμικού

Επιβλέπων: Παναγιώτης Βασιλειάδης, Αναπληρωτής Καθηγητής.

Η μελέτη της εξέλιξης της δομής των βάσεων δεδομένων, η οποία είναι γνωστή με τον όρο *εξέλιξη σχήματος*, είναι ιδιαίτερα σημαντική καθώς μπορεί να αποκαλύψει μοτίβα που θα βοηθήσουν τους διαχειριστές των βάσεων να αφιερώνουν λιγότερο χρόνο στην αύξηση της χωρητικότητας των παρεχόμενων πληροφοριών με τις λιγότερες πιθανές συνέπειες για τις εξαρτημένες εφαρμογές και να υλοποιούν όλες τις απαραίτητες εργασίες συντήρησης για να διατηρείται και να βελτιώνεται η απόδοση της βάσης.

Το βασικό, ερευνητικής φύσεως, ερώτημα που επιδιώκουμε να απαντήσουμε στην παρούσα εργασία μπορεί να διατυπωθεί ως εξής: *Υπάρχει κάποια σχέση μεταξύ της συσχέτισης των πινάκων με τα ξένα κλειδιά μιας βάσης δεδομένων και της εξέλιξής τους;* Για να απαντήσουμε στο συγκεκριμένο ερώτημα, χρησιμοποιούμε ένα μοντέλο που αναπαριστά κάθε έκδοση του σχήματος σαν έναν γράφο του οποίου οι κόμβοι και οι ακμές αντιστοιχούν στους πίνακες και τα ξένα κλειδιά του σχήματος, αντίστοιχα. Η ένωση αυτών των γράφων αντιστοιχεί στον *Διαχρονικό Γράφο*, ο οποίος αποτελείται από όλους τους πίνακες και όλα τα ξένα κλειδιά που εμφανίστηκαν σε τουλάχιστον μία έκδοση της ιστορίας του σχήματος. Επίσης, ορίζουμε 4 κατηγορίες για τους πίνακες, συγκεκριμένα τις *isolated*, *source*, *lookup* και *internal* με βάση τον συνδυασμό των έσω- και έξω-βαθμών τους στον Διαχρονικό Γράφο. Χαρακτηρίζουμε τις κατηγορίες αυτές με τον όρο *τοπολογικές*, καθώς περιγράφουν τη θέση των πινάκων στον Διαχρονικό Γράφο σε σχέση με τα ξένα κλειδιά τους. Στη συνέχεια, ταξινομούμε τους πίνακες στις τοπολογικές κατηγορίες και μελετάμε πώς η τοπολογία των πινάκων σχετίζεται με διάφορα χαρακτηριστικά της εξέλιξης τους, όπως η διάρκεια ζωής τους, η δραστηριότητα τους, η αλλαγή του μεγέθους τους, κ.α. Τα σχήματα που χρησιμοποιούμε στα πλαίσια της έρευνας μας προέρχονται από 5 σχεσιακές βάσεις δεδομένων που υποστηρίζουν συστήματα λογισμικού ανοιχτού κώδικα.

Τα σημαντικότερα αποτελέσματα της έρευνας μας, τα οποία επιβεβαιώνονται και από τα στατιστικά στοιχεία, είναι: (α) η σχέση μεταξύ των τοπολογικών κατηγοριών των πινάκων και της πιθανότητας εμφάνισης τους στην πρώτη έκδοση του σχήματος της βάσης τους και (β) η συσχέτιση της τοπολογίας των πινάκων με τη δραστηριότητα τους. Συγκεκριμένα, διαπιστώσαμε ότι όσο πιο σύνθετος τοπολογικά είναι ένας πίνακας τόσο πιο έντονη δραστηριότητα έχει και τόσο μεγαλύτερη είναι η πιθανότητα εμφάνισης του στην πρώτη έκδοση του σχήματος του.

Για να διευκολύνουμε την έρευνα μας, υλοποιήσαμε μία εκτεταμένη αναμόρφωση της αρχιτεκτονικής του λογισμικού Παρμενίδεια Αλήθεια, το οποίο οπτικοποιεί την εξέλιξη του σχήματος σχεσιακών βάσεων δεδομένων. Έχοντας εντοπίσει και προτεραιοποιήσει τα σχεδιαστικά ελαττώματα, εφαρμόσαμε μια σειρά από τροποποιήσεις στον πηγαίο κώδικα του εργαλείου με στόχο να αυξήσουμε τις δυνατότητες επέκτασης των λειτουργιών που προσφέρει το λογισμικό αυτό. Για να επιβεβαιώσουμε ότι οι αλλαγές που υλοποιήσαμε δεν έχουν επηρεάσει την αναμενόμενη συμπεριφορά του λογισμικού, δημιουργήσαμε ελέγχους μοναδιαίων ενοτήτων για κάθε ενότητα που είτε τροποποιήσαμε είτε προσθέσαμε. Τέλος, αξιολογούμε τις βελτιώσεις που επέφερε η διαδικασία αναμόρφωσης, συγκρίνοντας την ποιότητα της σχεδίασης του εργαλείου πριν και μετά την αναμόρφωση.

Συμπληρωματικά της αναμόρφωσης του λογισμικού, δημιουργήσαμε μία διαδικτυακή εφαρμογή που οπτικοποιεί την εξέλιξη του σχήματος σχεσιακών βάσεων δεδομένων και συνοψίζει τις βασικότερες πληροφορίες για την εξέλιξη των βάσεων.

CHAPTER 1.

INTRODUCTION

1.1 Scope

There is no doubt that the life cycle of a software product includes a series of changes that aim to either correct potential faults or extend its existing functionalities. Over the course of time, applications tend to increase the services they offer to their users and as consequence they are becoming more dependent upon their databases by retrieving more information from them. This entails a sequence of modifications to the database that alter its *internal structure* or its *schema* between successive versions. We use the term *schema evolution* to refer to these changes, which encompass tables/foreign keys insertions and removals as well as key and type updates.

The importance of studying schema evolution and understanding the mechanisms behind the necessity for changing database's structure can be realized if we consider that minor changes such as a foreign key removal or an attribute insertion can affect the surrounding applications leading to applications' failures or information loss. Identifying potential patterns in the evolution of databases' schemata can help administrators to maintain or develop databases in a way that eliminates the effects on the dependent applications and reduces the time and the effort they have to devote to apply the required modifications in the structure of the databases.

The so far limited number of the existing studies on the topic of relational databases' evolution can be attributed to the unavailability of a large number of open-source databases' schema histories that would allow us to establish solid conclusions on how schemata evolve over time and what are the factors that determine their evolution. It is worth mentioning that until 1993 there was no any in-depth study concerning schema evolution revealing a research gap in this topic. This gap was partially filled in the following years due to the presence of few open-source software projects that led to various works, which covered different aspects of the evolution, ranging from coarse-grained approximations that identify the effects of schema changes and propose methods for eliminating them to more fine-grained analyses that involve studying tables and foreign keys' evolution and determining which tables' properties are liable to affect tables' update activity.

Our approach on the topic of schema evolution is twofold, consisted of the *research* and the *tool support* parts. In the first part, we deal with the problem of schema evolution from a new perspective that takes into account *tables' involvement with foreign keys*, which means that we are mainly interested in understanding how the patterns of edges surrounding nodes in the *Diachronic Graph*, the graph whose nodes and edges represent databases' tables and foreign key constraints, respectively, is related to the evolution activity of databases' schemata. We should mention that in the context of this Thesis we use the term *topology* to describe tables' involvement with foreign keys in the Diachronic Graph. The second part of our work contains a principled refactoring process on a existing tool for the study of schema evolution and the utilization of this tool to construct a web application that can facilitate the works of research community on the topic of schema evolution.

In a nutshell, the research question that we attempt to answer in this thesis can be stated as follows: *"Is there a relationship between tables' topological categories and their evolution?"*

To answer this question, we utilize the schema histories of 5 databases supporting open-source projects from different domains. First, we study the distribution of the tables with respect to the combination of their in- and out-degrees in the Diachronic Graph to define the different topological categories to which we append the including tables. We identify four different categories in terms of tables' topology, which can be synopsized as follows:

- *Isolated* tables with no references from or to other tables.
- *Lookup* tables with only incoming references.
- *Source* tables with only outgoing references.

- *Internal* tables with both incoming and outgoing references.

Having determined the topological categories, we encounter the first problem arising from the so-called *change of category* phenomenon, which occurs when tables change category throughout their history. As a result, the tables are divided into those with a *single* topological label and those with *multiple* labels. A multi-labeling scheme does not facilitate our attempt to relate tables' topological labels to their evolution profile and for this reason we *manually* track tables that change category and assign a single label to them. The manual inspection of the label changes also helps us to determine a list of filters consisted of 6 rules that would automate the classification process of the multi-label tables by removing or ignoring bewildering parts of tables' lives that confuse the true nature of the tables. Although the misclassification rate between the two alternative processes is not high, *we adopt the labels derived from the manual process*, since it allows us to take into account the special features of the tables examined.

Assigning a single label to each table enables us to study whether tables' topological categories are related to various measures of their evolutionary activity, such as their lives' duration, their survival potential, their update activity etc. To assert whether the topology of the tables affects their evolution-related properties, we accompany the results derived from our study with statistical evidence by utilizing the Chi-square and Fisher tests.

The first question that we address is stated as follows: "*Is there a relationship between tables' topological categories and their duration?*" We classify tables in three categories with respect to their *normalized duration*, which is defined for a table as the ratio of the number of versions that the table exists in the dataset over the total number of versions of its dataset. Although we identify several duration-related patterns, the statistical tests we conducted to evaluate the differentiation of tables' duration due to their topological categories does not allow us to strongly support that there is a correlation between tables' topology and their lives' duration. The commonalities that we observe in the datasets examined are outlined in the following list:

- *Internal* and *lookup* tables tend to lives of *long* duration.
- *Isolated* tables avoid existing for a *long* period of time.

The second relationship that we are interested in is that between tables' topology and their survival potential. We describe a table as a "survivor" if it exists in the last known version of its dataset. The relevant research question that we attempt to answer can be stated as follows: "*Is there a relationship* *between tables' topological categories and their survival potential?"* The high survival rates, which surpass the 65% of the number of tables in all datasets, along with the results produced by the statistical tests indicate that is quite unlikely that tables' topology affect their survival potential. Nevertheless, we identify two interesting patterns summarized as follows:

- There exists a *monotone decrease* pattern in the percentages of the including "survivor" tables, starting from the *source* tables followed by *lookup* and ending with the *internal* tables.
- *Source* and *lookup* tables' survival rates follow the aggregate ones, while the survival potential of the *internal* tables is higher than the corresponding aggregate in all datasets.

Next, we examine whether the topology of the tables is somehow related to the originating version of their dataset's schema history. We can express the respective research question in the following way: *"How is the topological category of a table related to the probability of being born in the originating version of its dataset's schema history?"* The main findings of our study on this relationship are synopsized as listed below:

- *Internal* and *lookup* tables are more likely to be "born" in the originating version of their dataset's history, which means that it is not probable that these tables are introduced in the succeeding versions.
- *Isolated* and *source* tables tend to be born in versions that follow the originating one.
- The aforementioned results are in accordance with the *gravitation to rigidity* pattern, which suggests that dependency-magnet tables, like *internal* and *lookup*, are not prone to experience any kind of modifications in the later versions of database's schema. Thus, we presume that the early introduction of these tables is preferable in order to avoid changes caused by adding them in subsequent versions.

The update profile of the tables and its relationship with the topological categories is another issue that we study. The question that concerns us can be formulated in this way: *"Is there a relationship between the topological category of a table and its update activity?"* We classify tables with respect to their update activity in three categories, using the label *rigid* for those with no changes in their lives, the label *quiet* for tables with few changes that do not exceed the value of 5 and the label *active* for tables with more than 5 updates and with *Average Transitional Update* (ATU) greater than 0.1. The ATU represents the fraction of the number of changes a table experiences in its life over its duration. The main findings on the relationship between topological categories and tables' update profile can be listed as follows:

- *Isolated* and *source* tables are associated with *no* or *few* updates.
- *Lookup* tables experience either *few* or *many* changes.
- *Internal* tables are prone to undergo *many* updates.

We also examine the probability for a table with certain update activity to belong to a specific topological category. We outline the most interesting results in the subsequent list:

- *Rigid* tables are quite likely to be *source* in datasets where there is no strong presence of *isolated* tables, while in datasets with numerous *isolated* tables the *rigid* ones tend to be *isolated*.
- *Quiet* tables are likely to belong to the *source* category.
- *Active* tables tend to categories of high topological complexity.

Given the aforementioned findings as well as the results from the statistical tests, *we can claim that tables with different topological categories are subjects to different amounts of updates*.

The last relationship we study is that between the tables' topological categories and their size change, meaning the change of their size between their first and last known versions. The research question we attempt to answer is expressed in this way: "*How is the topological category of a table related to its size change?*" We categorize tables with respect to the scale of their size change in three categories, with the label *scale down* denoting a reduction in a table's size, the label *steady* representing tables with unchanged sizes and the label *scale up* indicating an expansion in tables' sizes. Although the statistical evidence is not adequate enough to support a correlation between topological categories and tables' size change, we identify the following behaviors:

- The majority of the *isolated* and *source* tables remain *steady*.
- *Lookup* and *internal* tables tend to increase their size.

The second topic of our thesis concerns the *refactoring* process applied in Parmenidian Truth project, a tool that visualizes the evolution of relational databases' schemata. The main reason for improving the design of this tool is that we utilize its functionalities for creating the web application presented in

Chapter 5, so a series of refactoring actions would facilitate the introduction of new functionalities required by our application. Using the *Unified Modeling Language* (UML) along with the Classes Collaborations Responsibilities (CRC) method, we are able to identify violations or the absence of design principles that would complicate our effort to add new functionalities or make use of the existing ones provided by the Parmenidian Truth tool. The defects we inspect are summarized as follows: (i) Package level issues (ii) God classes (iii) Lack of APIs (iv) Duplicated code (v) Misplaced methods (vi) Redundant components (vii) Convention violations

To deal with each of the above mentioned defects we apply a series of modifications in the source code of the tool taking into consideration and complying with the proposed, in each case, design principles and patterns. Next, we create a unit test for each class we added or modified to confirm that our alterations have not affected the expected behavior of the tool. Finally, we conduct a thorough evaluation of the quality of the resulting source code after the refactoring actions we applied in order to assess and quantify the enhancements achieved in design level.

In a nutshell, the main improvements achieved via the refactoring process can be outlined as follows:

- We have increased tool's expandability and immunity to changes by introducing a set of APIs
- We have eliminated duplicated code by applying the recommended *template method* design pattern
- We have increased cohesion of methods by moving misplaced methods to new classes
- We have removed redundant components that increase code complexity
- We have verified the correctness of the proposed modifications by creating unit tests for classes that we either added or modified

The third part of this thesis describes the structure of a web application we create to visualize the schema evolution of relational databases. Our prime motive for creating this application was to provide the entire research community with a tool that can facilitate their work on the topic of schema evolution. We utilize the refactored version of the Parmenidian Truth tool and its functionalities to upload all the necessary information on the server and exploit it each time a client's request is submitted. The main functionalities provided by our application are summarized as follows:

- An overview on the distribution of the tables with respect to several properties, like their update activity, birth version, etc.
- Visualization of tables and foreign keys' evolution as well as the depiction of the four evolution-related patterns presented in [VaZS15].
- Visualization of the Diachronic Graph and the intermediate versions. We also provide users with the capability of selecting nodes' classification criterion and setting nodes' radius based on different tables' properties.

To sum up, the main contributions of this thesis are synopsized in the next list:

- A thorough study on the relationship between tables' topology and their evolution.
- An extensive restructuring of the Parmenidian Truth tool's architecture and an in-depth evaluation of the refactoring process.
- A web application that facilitates the visualization of databases' schemata evolution.

1.2 Roadmap

The contents of this thesis can be summarized as follows. In Chapter 2, we highlight the most significant contributions on the topic of schema evolution and explain how our work differentiates from the state of the art. In Chapter 3, we present and assess the modifications we applied to Parmenidian Truth tool to improve its architecture, aiming at utilizing it to create a web application that visualizes schema evolution. Chapter 4 contains our study on how tables' topological labels are related to tables' evolution. In Chapter 5, we present our web application that facilitates the study on the evolution of databases' schema histories via visualizing various known patterns and providing the corresponding quantitative information. In Chapter 6, we outline the most important conclusions of our work and highlight open issues for future research.

CHAPTER 2.

RELATED WORK

In this Chapter, we present the state of the art in the related literature on the topic of this thesis so as to highlight the growing interest for schema evolution and what has been achieved over the years. In the second section, there appears a brief comparison of our work to the case studies of the first section, demonstrating how our work diverges from the previous ones and contributes to broadening our knowledge over the subject of schema evolution.

2.1 Case Studies of Schema Evolution

One of the earliest works in the area of schema evolution was implemented in 1993 by [Sjøb93], who studied the evolution of a database for a period of 18 months and demonstrated the need for the development of a change management tool. The main findings of his work can be outlined as follows:

- Every relation of the database has been modified during the period of the study.
- More additions than deletions appeared in the early phases of the databases' lives, in contrast to the operational period in which the additions and deletions were in balance.

- There was a 139% increase in the number of relations and a 274% growth in the number of fields, concerning the period of examination.

In 2002, Amela Karahasanovic [Kara02] presented a tool for tracing the effects of schema changes in applications developed in object – oriented systems. This tool, namely Schema Evolution Management Tool (SEMT), receives as input the source files of a schema, identifies the modules of the schema and their relationships, creates a graph – based representation with the nodes corresponding to schema's modules and the edges to the relationships between the modules and predicts the impacts of changes applied on the schema. The evaluation of the tool was carried out by conducting an experiment in which two groups of students were asked to apply changes in the schema of a library application and subsequently identify the effects of the changes by using SEMT. Each group used a different version of the SEMT tool, with the first group utilizing a version that recognizes the impacts of a change at a fine – grained level and the second one exploiting a version that determines the affected modules at a coarse – grained level. The results of the experiment, which consisted of the time required to complete the impact analysis, the correctness of the answers and the user satisfaction, are the following:

- The time required to complete the experiment was 6 minutes shorter for the group utilizing the low – level version of the SEMT tool.
- Students using the fine – grained version of the tool committed fewer errors in their effort to discover the parts of the schema that were affected by a change.
- The score regarding users' satisfaction and viewpoint about SEMT's efficiency was high within the two groups.

In 2008, Carlo Curino, Hyun Moon and Carlo Zaniolo [CuMZ08] introduced a set of Schema Modification Operators (SMOs) to facilitate the evaluation of the effects of the schema changes and minimize the maintenance costs involved in terms of time and effort required to identify the parts affected. Each of the SMOs corresponds to a function whose parameters are a relational schema and the underlying database and its output is a modified version of the initial schema and a migrated version of the database. In this context, they developed the Panta Rhei Information & Schema Manager (PRISM) system, which automates the completion of tasks associated with schema evolution such as query translation, data migration and documentation of the changes. As for the assessment of the PRISM system, they exploited the schema evolution history of Wikipedia to quantify the efficiency of the PRISM in terms of the proportion of the evolution steps automated by the system and the percentage of the queries that were compatible with the new schema version. The results obtained from this experimental evaluation are summarized in the following list:

- In 97.2% of the evolution steps the PRISM system was capable of adjusting queries to the new schema version in an automatic way.
- 74% of the queries were operational after the required modifications applied by the PRISM system, in contrast to the 16% of the compatible queries in case of no changes would have been introduced.
- In 12% of the queries altered there appeared a gain of 35% in terms of the execution time in favor of the manually modified queries and that was attributed to the fact that the PRISM system was incapable of identifying integrity constraints.

In 2008, Carlo Curino, Hyun Moon, Letizia Tanca and Carlo Zaniolo [CMTZ08] made a thorough analysis of the evolution history of the Wikipedia and its schema, covering a period of approximately 4.5 years. Acknowledging the profound impact schema changes have on the applications accessing a database, they initially performed a macro – and micro – classification of the schema changes and then they evaluated the effect of the changes on applications by quantifying the success rate of the queries execution among different schema versions. The following list puts in a nutshell the key findings of their study.

- The majority of the evolution steps (nearly 55%) included actual schema changes and more than 40% of the steps concerned key/index adaptations.
- The micro – classification of the schema changes revealed an equilibrium between additions and deletions of tables and attributes, which signifies the intention to preserve database’s contents.
- Only 22% of the queries of previous versions are functional in subsequent versions.

Shengfeng Wu and Iulian Neamtiu [WuNe11] focused their research on schema evolution of embedded databases and proposed a system for the automatic retrieval of the source code, the extraction of the embedded databases and the computation of the schema evolution. They employed 4 well – known applications containing embedded databases and studied its evolution within an 18 – year period. The main findings of this study are condensed as follows:

- The high frequency of tables and attributes deletions indicate that embedded databases are more prone to restructuring growth rather than a continuous one.
- The early periods in schemas' lives are related to higher number of changes as opposed to the latter versions which include few modifications.
- The overall change rate for the embedded databases tends to be lower than that of the enterprise – class databases.

In 2012, G. Papastefanatos, P. Vassiliadis, A. Simitsis and Y. Vassiliou [PVSV12] presented their work about the impact of evolution events on the Extract – Transform – Load (ETL) workflows and proposed a set of graph – theoretic metrics for the assessment of the quality of ETL designs. First, they develop a graph – based model to represent the modules of an environment and following that they analyze its structure to determine the extent to which evolution events can affect environment's components. The evolution graph representing the parts of an ETL system and its relationships was analyzed in two levels, specifically a fine – grained level where node properties are examined as potential predictors of node's vulnerability to evolution actions and a coarse – grained level concerning only relations, views and queries. The proposed metric suite used for the structure analysis of the evolution graph includes degree – based metrics, such as simple or transitive degrees of nodes or modules, indicating the level of dependencies among nodes and modules and entropy – related metrics which signifies the possibility for a node to be affected by a random evolution event. The evaluation of the proposed metrics was implemented by exploiting a software tool, namely Hecataeus, which in this study analyses 7 real – world ETL scenarios for 6 months. The most important observations derived from this experimental evaluation are synopsized as follows:

- The schema size of a system is a crucial factor behind system's vulnerability to evolution events, that is to say that tables with many attributes are more likely to be affected and affect the corresponding work flows.
- Out – degree of nodes and modules are the most adequate predictors for the evolution of all module types.

- In cases where the previous metrics fail, the out – transitive degree and the entropy – related metrics may operate as better predictors for the impact of evolution on ETL workflows.

In [QiLS13], authors studied the co – evolution of database schemata and the code of the related applications in 10 open – source projects. The main research questions addressed concerned the frequency and amount of schema evolution, the distribution of the schema change types within databases' lives and the evaluation of the impact of schema changes on the corresponding application code. They classified atomic schema changes into 24 categories, each of which belongs to one out of 6 high – level schema change types, so as to discriminate the dominant types of modifications and assess the effect of each type to the surrounding applications. The following list includes the main results of this study.

- The frequency of schema modifications is high, with the average number of atomic changes to approximate the value of 90 in a year.
- The growth rate of tables in 60% of the databases exceeds 100% as it is the case for the change rate in 90% of the projects examined.
- In all but 3 projects their schema size approaches the 60% of their maximum value within the first 20% of their lifetimes.
- Regarding the distribution of schema change types in databases' lives, it appears that transformations, structure refactorings and data quality refactorings are the most common categories of changes accounting for 80% of schema changes in all projects and 95% in 7 of them.
- Additions of tables/columns and datatype changes are the most frequent changes at the low – level of change categories.
- Each atomic change affects approximately from 10 to 100 Lines of Code on average. At a coarse – grained level, transformations and structure refactorings are responsible for the most of the alterations required in the source code of the surrounding applications.

In 2015, A. Cleve et al. [CGMM+15] published their findings on the adequacy of using the database evolution history as an effective tool in reverse engineering procedures. Specifically, they studied the evolution history of a medical record application seeking for valuable information that would assist system's extendibility capacities in order to comply with new requirements. To achieve their main goal, they developed a set of tools for retrieving, analyzing and visualizing the schema versions of the database accessed by the aforementioned application. The main results of their work are highlighted in the following list.

- The number of tables appears to be increasing from the beginning till the end of the period examined, unveiling an obvious reluctance to remove tables.
- The trend in the evolution of attributes approximates that of tables.
- The addition of large tables spans the whole life of the system under examination.
- The update activity in the database schema is far from being intense with the majority of tables experiencing less than 4 changes in their lives.

P. Vassiliadis, A. V. Zarras and I. Skoulis [VaZS17] performed an in – depth analysis on the schema evolution of 8 databases aiming at perceiving how individual tables evolve and studying the impact of various tables properties on tables' lives. Specifically, they investigate whether or not properties such as schema size, birth/removal versions are associated with evolution – related features, for instance table's update activity, duration, survival profile. The key findings of their study are outlined in the subsequent list.

- Wide tables, these are tables born with more than 10 attributes, are more likely to survive, in other words to exist in the last schema version. With the exception of 2 datasets, the percentage of those tables exceeds 85% in all cases.
- In 50% of the datasets the portion of wide tables that were born early, that is to say in the lowest 33% of versions, and finally survive, surpasses 70%.
- Approximately 70% of tables of a database resides within the 10x10 box, meaning that the number of attributes at the birth version does not exceed the value of 10 and the number of updates a table undergoes throughout its life is less than 10.
- As for the relationship of tables' duration with their update profile, they introduce the "inverse Γ" pattern which indicates that short – lived tables are related with a small amount of changes, tables of medium duration undergo a small or medium number of changes and long – lived tables are subjects to all kinds of updates.

- More than 75% of active tables, those are tables having an Average Transitional amount of Update (ATU) greater than 0.1 and experiencing more than 5 changes in their lifetime, are born early.
- Apart from 2 datasets, the fraction of active tables that survive is greater than 70%.
- The probability for an active, long – lived table to survive is 100%, as it is the probability for active, long – lived survivors to have been born early.
- In 6 out of 8 datasets, the percentage of removed tables that experience few updates exceeds the value of 85%.
- With 1 exception, the fraction of removed tables that were born early is greater than 70%.
- Removed tables that are short – lived accounts for more than 75% of the total number of "dead" tables in all but three datasets.

In 2017, P. Vassiliadis et al. [VKZZ17] studied how foreign keys evolve in the context of schema evolution of relational databases. Recognizing the impact of the schema evolution on the smooth operation of the surrounding applications and the importance of predicting forthcoming schema changes for the maintenance process, they opted for six open – source databases derived from different domains and included foreign key constraints. First, they represented each version of the schema of a database as a graph with relations as nodes and foreign keys as graph's edges and then detected a set of changes between subsequent versions by utilizing the Parmenidian Truth tool that models, visualizes and quantifies schema evolution of a database. The main findings of this work are summarized as follows:

- The growth of the schemata is continuous including alternating phases of concentrated modifications and of few or zero changes.
- In most datasets, there seems that foreign key constraints are rare and in some cases their existence appears to be unwelcome.
- The evolution of foreign keys does not always follow that of tables.
- The heartbeat of foreign key changes is mostly rare and small in volume.

2.2 Comparison to the State of the Art

In the previous section we attempted to give a synopsis of the different approaches to the matter of schema evolution and the most significant contributions of each work towards understanding the mechanisms that determine how schemata evolve in terms of their main components including tables and foreign key constraints and whether a set of tables' properties such as their size, duration, update activity, etc. is likely to affect their evolution. To the best of our knowledge, this is the first study that examines the role of tables' topology in the evolution process, which expressed in a different way means that we are going to focus our research on how and to what extent the "neighborhood" of a table affects its life in terms of its survival likelihood, its update profile or the duration of its life. Prior to studying the relationship between tables' evolution and their topological labels, we propose a set of rules for classifying tables into topological categories taking into account the changes of the corresponding labels throughout tables' life cycles.

CHAPTER 3.

REFACTORING PROCESS

In this chapter, we present the set of modifications we applied to Parmenidian Truth tool in order to improve its design and facilitate any attempt to extend its functionalities. Firstly, we explain why refactoring is required in the context of the current thesis and we show the initial design along with the corresponding defects. Next, we describe a series of refactoring actions applied aiming at eliminating design violations, mention the tests conducted to ensure that our changes did not affect the expecting behavior of the tool and present the design ensued after our alterations. Finally, we assess the benefits of the refactoring process.

The *Parmenidian Truth* tool visualizes the evolution of relational databases' schemata. It takes as input a set of data definition files that contain the history of a database and produces as output the *Diachronic Graph*, a graph whose nodes correspond to the tables that have existed in database's history for at least one version and edges model the foreign key constraints that have identified between the tables for at least one version. Apart from the Diachronic Graph, the Parmenidian Truth tool produces a PowerPoint presentation, where each slide illustrates a graph modeling of each version with the including tables depicted as nodes and the foreign keys as edges. The graph representation of a database's schema history was introduced in [VKZZ17] facilitating the correlation of graph-related metrics with evolution-related features. In this context, this tool also computes a set of graph-based measures for each version as well as for the entire history of the database.

3.1 Aim of Refactoring

One of the main objectives of this master thesis is to utilize the functionalities provided by the Parmenidian Truth tool. The fulfillment of new requirements and their adaptation to the existing code entail the understanding of tool's design and the evaluation of its quality. In a first step, we have to obtain an insight of Parmenidian Truth's structure disclosing either potential violations or lack of design principles, which might exacerbate the extension process and complicate forthcoming maintenance efforts. In a next step, we attempt to apply a series of modifications to source code in order to improve the design of the software in a way that will favor the extensibility and maintainability of the tool. This process is known with the term refactoring, which is explained in more detail in the following paragraph.

Terminology. *Refactoring* is used to describe a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [FBB+99].

The rationale behind the necessity of this process derives from the fact that we inherited the source code of the tool, so understanding its design and determining the margins for improvements is considered to be of great importance for the subsequent process of adding new features to the tool and exploiting them in the application presented in Chapter 5.

3.2 Initial Architecture and Design

In this subsection, we discuss the architecture of Parmenidian Truth before our refactoring took place. At first, we utilize diagram generators that offer graphical representations of the subject system at high – level, presenting the dependencies of its packages and also at package – level, revealing the relationships between the entities included.

3.2.1 Package Diagram

Figure 3.2 depicts the initial package diagram of Parmenidian Truth system before the refactoring. Each package represents a subsystem that offers a unique functionality required by the system in order to fulfill the requirements that this tool satisfies. As mentioned before, Parmenidian Truth's main functionality is the visualization of a database's schema as a PowerPoint presentation, so its subsystems are expected to cooperate in a way that this functionality is provided.

Figure 3.1 summarizes the functionalities provided by each subsystem.

Subsystem	Functionality
gui	Contains graphical interface – related classes
core	Operating as manager of the use cases system performs
export	Includes the classes that offer export – related operations
model.Loader	Organizing data using externalTools subsystem
model	Contains domain classes of the system
externalTools	Consists of Hecate tool's classes that parse SQL files
parmenidianEnumerations	Comprises useful enumerations

Figure 3.1 Functionalities of Subsystems

It is important to clarify that the *externalTools* package consists of classes of the Hecate tool, which is a different system and for this reason it was not modified during the refactoring process.

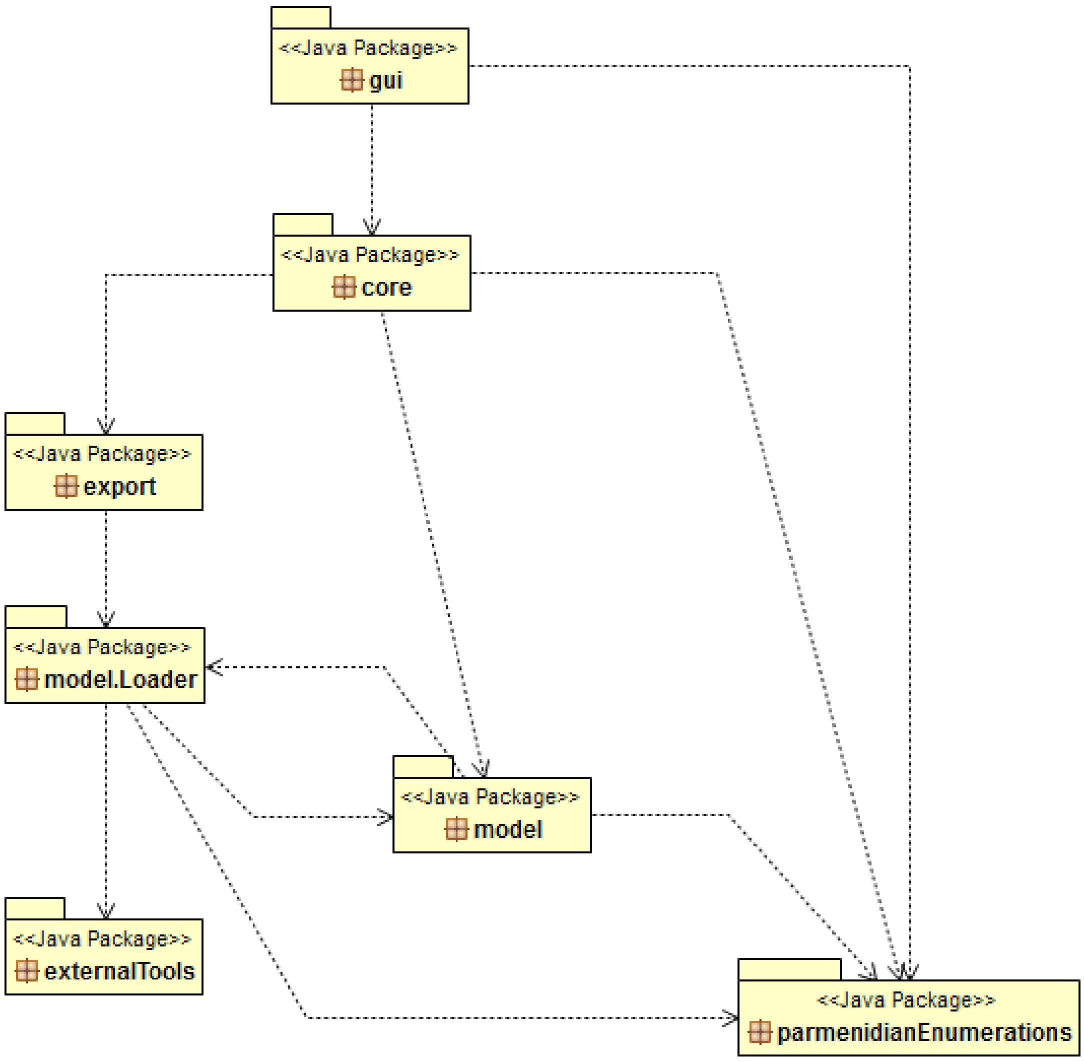
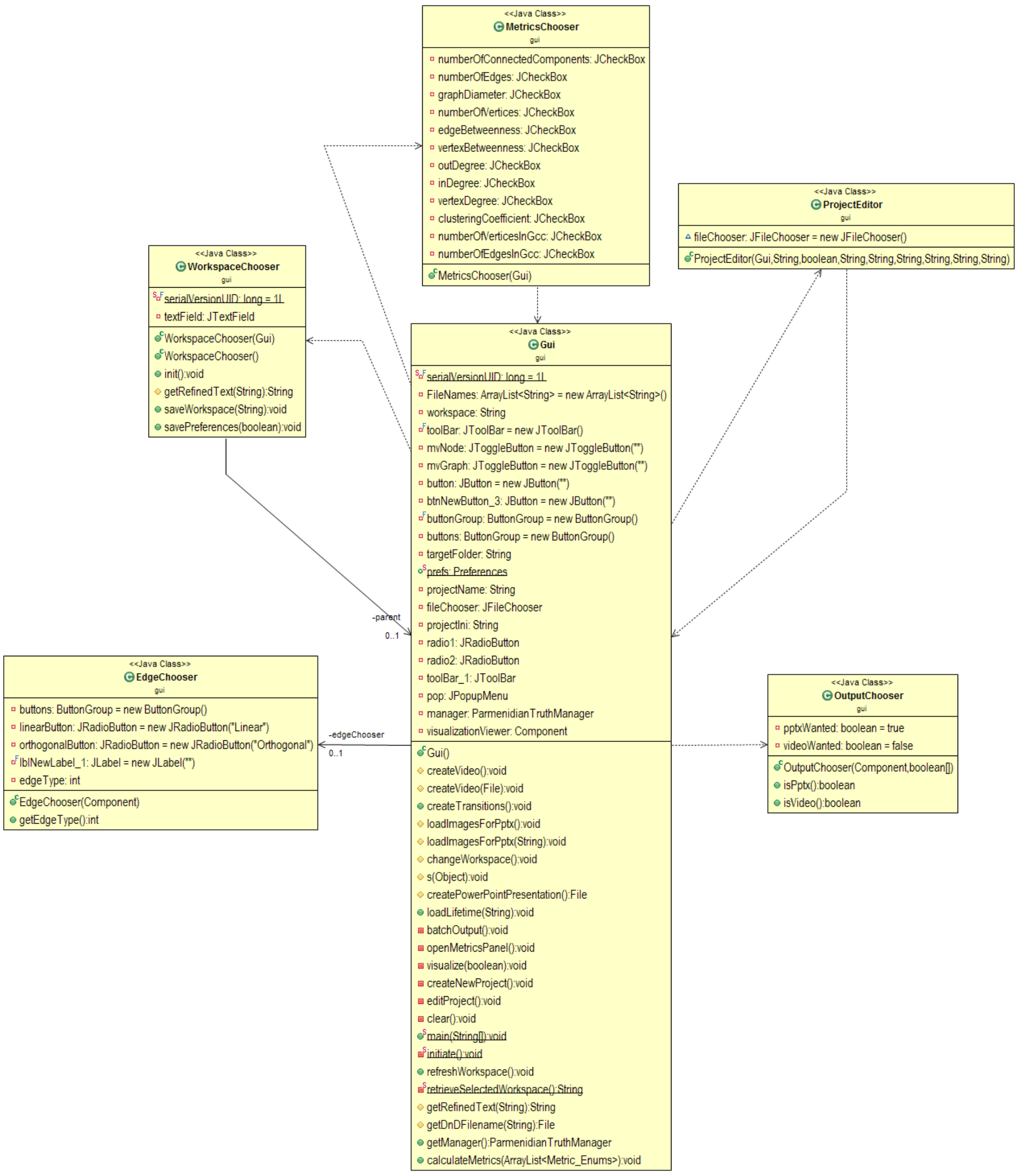


Figure 3.2 Initial Package Diagram of Parmenidian Truth

3.2.2 Class Diagrams

As mentioned before, each subsystem is supposed to offer a functionality that derives from the cooperation of its components. This means that the classes of each package are supposed to be strongly correlated to one another, aiming at serving a single purpose. The class diagrams of this subsection show the associations between the elements of each package giving a sense of the degree of cohesion within it.

Figure 3.3 corresponds to the class diagram of the *gui* package.



<<Java Class>>
MetricsChooser
gui
numberOfConnectedComponents: JCheckBox
numberOfEdges: JCheckBox
graphDiameter: JCheckBox
numberOfVertices: JCheckBox
edgeBetweenness: JCheckBox
vertexBetweenness: JCheckBox
outDegree: JCheckBox
inDegree: JCheckBox
vertexDegree: JCheckBox
clusteringCoefficient: JCheckBox
numberOfVerticesInGcc: JCheckBox
numberOfEdgesInGcc: JCheckBox
MetricsChooser(Gui)
<<Java Class>>
ProjectEditor
gui
fileChooser: JFileChooser = new JFileChooser()
ProjectEditor(Gui,String,boolean,String,String,String,String,String,String)
<<Java Class>>
WorkspaceChooser
gui
serialVersionUID: long = 1L
textField: JTextField
WorkspaceChooser(Gui)
WorkspaceChooser()
init():void
getRefinedText(String):String
saveWorkspace(String):void
savePreferences(boolean):void
<<Java Class>>
Gui
gui
serialVersionUID: long = 1L
FileNames: ArrayList<String> = new ArrayList<String>()
workspace: String
toolBar: JToolBar = new JToolBar()
mvNode: JToggleButton = new JToggleButton("")
mvGraph: JToggleButton = new JToggleButton("")
button: JButton = new JButton("")
btnNewButton_3: JButton = new JButton("")
buttonGroup: ButtonGroup = new ButtonGroup()
buttons: ButtonGroup = new ButtonGroup()
targetFolder: String
prefs: Preferences
projectName: String
fileChooser: JFileChooser
projectIni: String
radio1: JRadioButton
radio2: JRadioButton
toolBar_1: JToolBar
pop: JPopupMenu
manager: ParmenidianTruthManager
visualizationViewer: Component
Gui()
createVideo():void
createVideo(File):void
createTransitions():void
loadImagesForPptx():void
loadImagesForPptx(String):void
changeWorkspace():void
s(Object):void
createPowerPointPresentation():File
loadLifetime(String):void
batchOutput():void
openMetricsPanel():void
visualize(boolean):void
createNewProject():void
editProject():void
clear():void
main(String[]):void
initiate():void
refreshWorkspace():void
retrieveSelectedWorkspace():String
getRefinedText(String):String
getDnDFilename(String):File
getManager():ParmenidianTruthManager
calculateMetrics(ArrayList<Metric_Enums>):void
-parent
0..1
<<Java Class>>
EdgeChooser
gui
buttons: ButtonGroup = new ButtonGroup()
linearButton: JRadioButton = new JRadioButton("Linear")
orthogonalButton: JRadioButton = new JRadioButton("Orthogonal")
lblNewLabel_1: JLabel = new JLabel("")
edgeType: int
EdgeChooser(Component)
getEdgeType():int
-edgeChooser
0..1
<<Java Class>>
OutputChooser
gui
pptxWanted: boolean = true
videoWanted: boolean = false
OutputChooser(Component,boolean[])
isPptx():boolean
isVideo():boolean

Figure 3.3 Class Diagram of the Gui Package

Figure 3.4 depicts the class diagram of the *core* package.

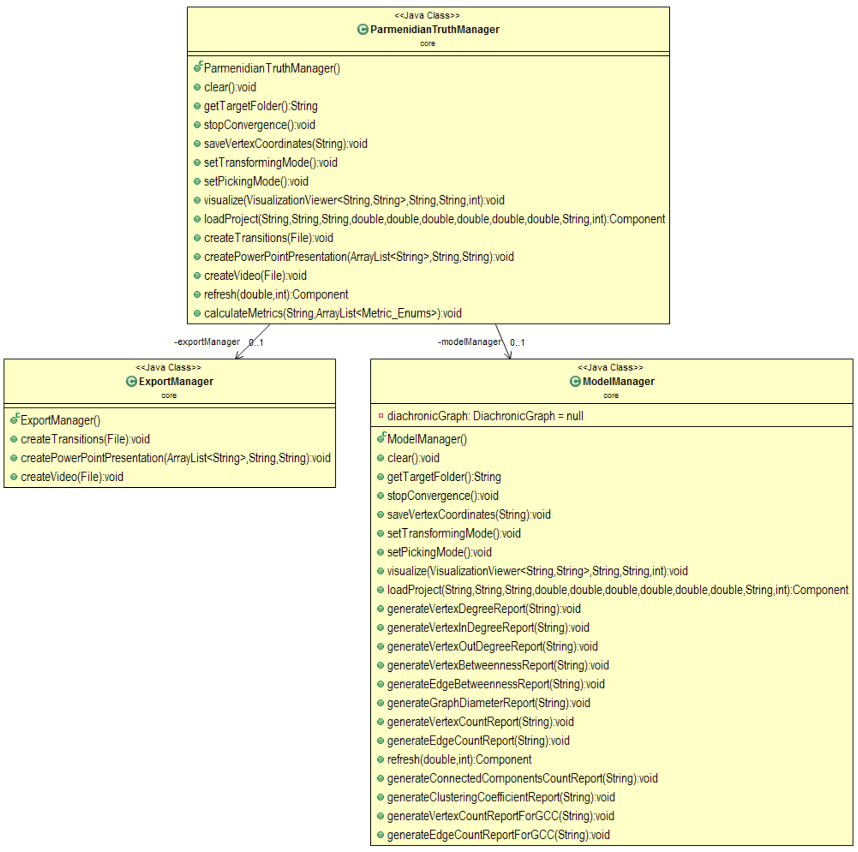

<<Java Class>>
ParmenidianTruthManager
core
ParmenidianTruthManager()
clear():void
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
createTransitions(File):void
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
refresh(double,int):Component
calculateMetrics(String,ArrayList<Metric_Enums>):void
-exportManager
0..1
-modelManager
0..1
<<Java Class>>
ExportManager
core
ExportManager()
createTransitions(File):void
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
<<Java Class>>
ModelManager
core
diachronicGraph: DiachronicGraph = null
ModelManager()
clear():void
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
generateVertexDegreeReport(String):void
generateVertexInDegreeReport(String):void
generateVertexOutDegreeReport(String):void
generateVertexBetweennessReport(String):void
generateEdgeBetweennessReport(String):void
generateGraphDiameterReport(String):void
generateVertexCountReport(String):void
generateEdgeCountReport(String):void
refresh(double,int):Component
generateConnectedComponentsCountReport(String):void
generateClusteringCoefficientReport(String):void
generateVertexCountReportForGCC(String):void
generateEdgeCountReportForGCC(String):void

Figure 3.4 Class Diagram of the Core Package

Figure 3.5 shows the classes included in the *export* package.

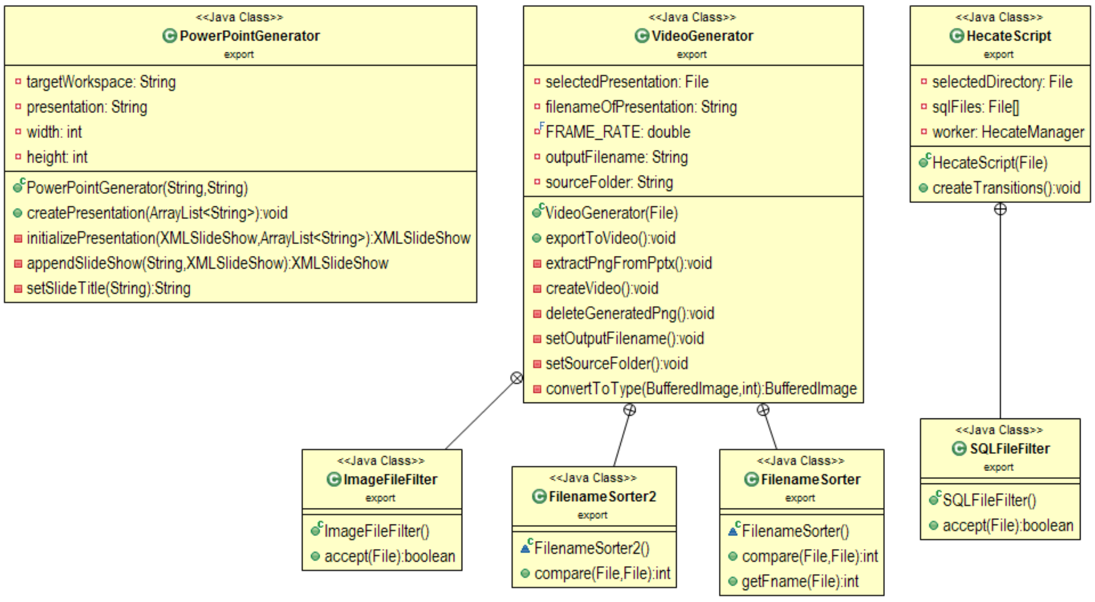
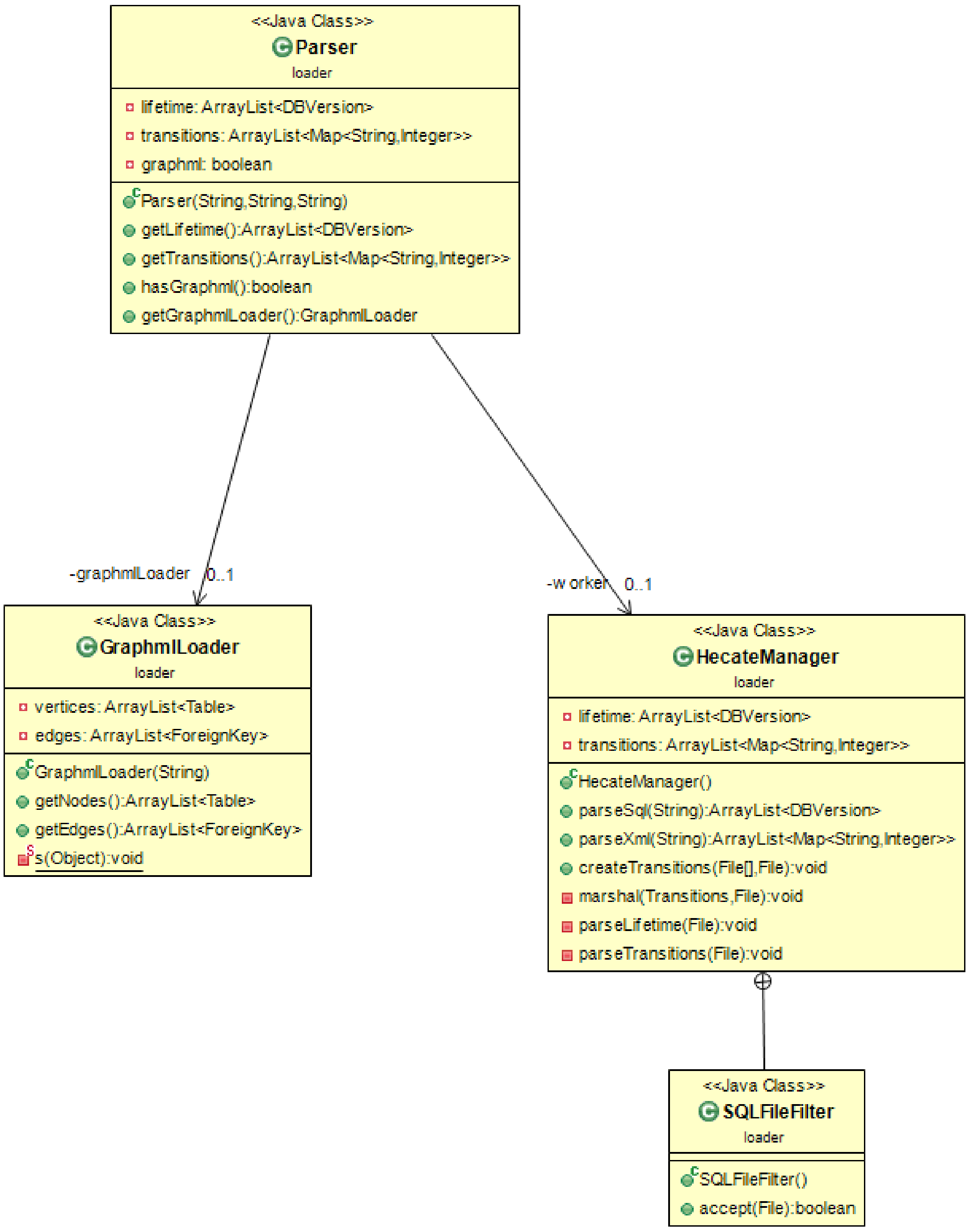


Figure 3.5 Class Diagram of the Export Package

Figure 3.6 presents the classes the *model.Loader* package consists of.



<<Java Class>>
Parser
loader
lifetime: ArrayList<DBVersion>
transitions: ArrayList<Map<String,Integer>>
graphml: boolean
Parser(String,String,String)
getLifetime():ArrayList<DBVersion>
getTransitions():ArrayList<Map<String,Integer>>
hasGraphml():boolean
getGraphmlLoader():GraphmlLoader
-graphmlLoader
0..1
-worker
0..1
<<Java Class>>
GraphmlLoader
loader
vertices: ArrayList<Table>
edges: ArrayList<ForeignKey>
GraphmlLoader(String)
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
s(Object):void
<<Java Class>>
HecateManager
loader
lifetime: ArrayList<DBVersion>
transitions: ArrayList<Map<String,Integer>>
HecateManager()
parseSql(String):ArrayList<DBVersion>
parseXml(String):ArrayList<Map<String,Integer>>
createTransitions(File[],File):void
marshal(Transitions,File):void
parseLifetime(File):void
parseTransitions(File):void
<<Java Class>>
SQLFileFilter
loader
SQLFileFilter()
accept(File):boolean

Figure 3.6 Class Diagram of the Model.Loader Package

Class diagram of the *model* package is shown in Figure 3.7.

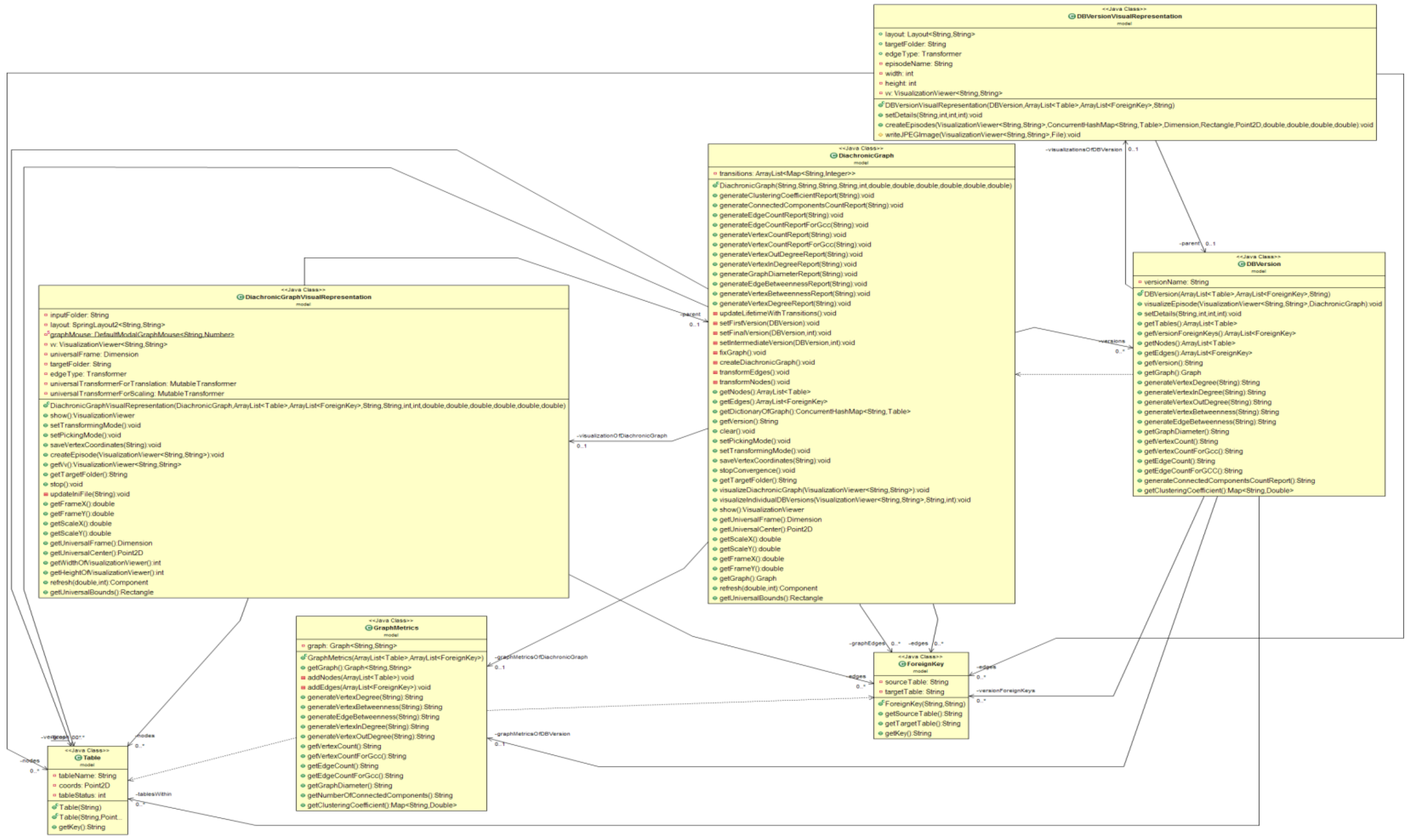
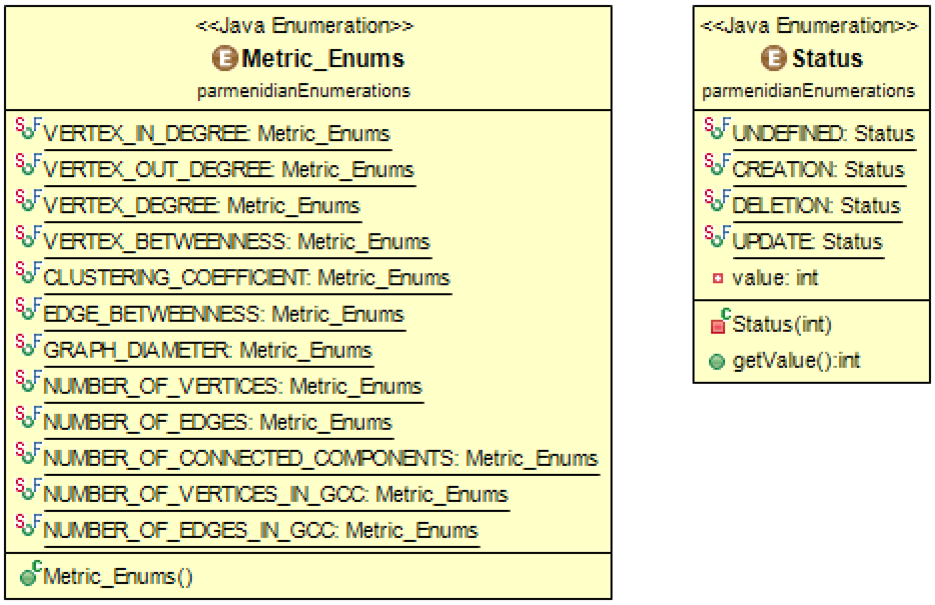

<<Java Class>>
DBVersionVisualRepresentation
model
layout: Layout<String,String>
targetFolder: String
edgeType: Transformer
episodeName: String
width: int
height: int
vv: VisualizationViewer<String,String>
DBVersionVisualRepresentation(DBVersion,ArrayList<Table>,ArrayList<ForeignKey>,String)
setDetails(String,int,int,int):void
createEpisodes(VisualizationViewer<String,String>,ConcurrentHashMap<String,Table>,Dimension,Rectangle,Point2D,double,double,double,double):void
writeJPEGImage(VisualizationViewer<String,String>,File):void
-visualizationsOfDBVersion
0..1
<<Java Class>>
DiachronicGraph
model
transitions: ArrayList<Map<String,Integer>>
DiachronicGraph(String,String,String,String,int,double,double,double,double,double,double)
generateClusteringCoefficientReport(String):void
generateConnectedComponentsCountReport(String):void
generateEdgeCountReport(String):void
generateEdgeCountReportForGcc(String):void
generateVertexCountReport(String):void
generateVertexCountReportForGcc(String):void
generateVertexOutDegreeReport(String):void
generateVertexInDegreeReport(String):void
generateGraphDiameterReport(String):void
generateEdgeBetweennessReport(String):void
generateVertexBetweennessReport(String):void
generateVertexDegreeReport(String):void
updateLifetimeWithTransitions():void
setFirstVersion(DBVersion):void
setFinalVersion(DBVersion,int):void
setIntermediateVersion(DBVersion,int):void
fixGraph():void
createDiachronicGraph():void
transformEdges():void
transformNodes():void
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
getDictionaryOfGraph():ConcurrentHashMap<String,Table>
getVersion():String
clear():void
setPickingMode():void
setTransformingMode():void
saveVertexCoordinates(String):void
stopConvergence():void
getTargetFolder():String
visualizeDiachronicGraph(VisualizationViewer<String,String>):void
visualizeIndividualDBVersions(VisualizationViewer<String,String>,String,int):void
show():VisualizationViewer
getUniversalFrame():Dimension
getUniversalCenter():Point2D
getScaleX():double
getScaleY():double
getFrameX():double
getFrameY():double
getGraph():Graph
refresh(double,int):Component
getUniversalBounds():Rectangle
-parent
0..1
-parent
0..1
<<Java Class>>
DBVersion
model
versionName: String
DBVersion(ArrayList<Table>,ArrayList<ForeignKey>,String)
visualizeEpisode(VisualizationViewer<String,String>,DiachronicGraph):void
setDetails(String,int,int,int):void
getTables():ArrayList<Table>
getVersionForeignKeys():ArrayList<ForeignKey>
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
getVersion():String
getGraph():Graph
generateVertexDegree(String):String
generateVertexInDegree(String):String
generateVertexOutDegree(String):String
generateVertexBetweenness(String):String
generateEdgeBetweenness(String):String
getGraphDiameter():String
getVertexCount():String
getVertexCountForGcc():String
getEdgeCount():String
getEdgeCountForGCC():String
generateConnectedComponentsCountReport():String
getClusteringCoefficient():Map<String,Double>
-versions
0..*
<<Java Class>>
DiachronicGraphVisualRepresentation
model
inputFolder: String
layout: SpringLayout2<String,String>
graphMouse: DefaultModalGraphMouse<String,Number>
vv: VisualizationViewer<String,String>
universalFrame: Dimension
targetFolder: String
edgeType: Transformer
universalTransformerForTranslation: MutableTransformer
universalTransformerForScaling: MutableTransformer
DiachronicGraphVisualRepresentation(DiachronicGraph,ArrayList<Table>,ArrayList<ForeignKey>,String,String,int,int,double,double,double,double,double,double)
show():VisualizationViewer
setTransformingMode():void
setPickingMode():void
saveVertexCoordinates(String):void
createEpisode(VisualizationViewer<String,String>):void
getVv():VisualizationViewer<String,String>
getTargetFolder():String
stop():void
updateIniFile(String):void
getFrameX():double
getFrameY():double
getScaleX():double
getScaleY():double
getUniversalFrame():Dimension
getUniversalCenter():Point2D
getWidthOfVisualizationViewer():int
getHeightOfVisualizationViewer():int
refresh(double,int):Component
getUniversalBounds():Rectangle
-visualizationOfDiachronicGraph
0..1
<<Java Class>>
GraphMetrics
model
graph: Graph<String,String>
GraphMetrics(ArrayList<Table>,ArrayList<ForeignKey>)
getGraph():Graph<String,String>
addNodes(ArrayList<Table>):void
addEdges(ArrayList<ForeignKey>):void
generateVertexDegree(String):String
generateVertexBetweenness(String):String
generateEdgeBetweenness(String):String
generateVertexInDegree(String):String
generateVertexOutDegree(String):String
getVertexCount():String
getVertexCountForGcc():String
getEdgeCount():String
getEdgeCountForGcc():String
getGraphDiameter():String
getNumberOfConnectedComponents():String
getClusteringCoefficient():Map<String,Double>
-graphMetricsOfDiachronicGraph
0..1
-graphMetricsOfDBVersion
0..1
-graphEdges
0..*
-edges
0..*
<<Java Class>>
ForeignKey
model
sourceTable: String
targetTable: String
ForeignKey(String,String)
getSourceTable():String
getTargetTable():String
getKey():String
-edges
0..*
-edges
0..*
-versionForeignKeys
0..*
<<Java Class>>
Table
model
tableName: String
coords: Point2D
tableStatus: int
Table(String)
Table(String,Point...
getKey():String
-nodes
0..*
-nodes
0..*
-tablesWithin
0..*

Figure 3.7 Class Diagram of the Model Package

Class diagram of the *parmenidianEnumerations* package is depicted in Figure 3.8.



<<Java Enumeration>>
Metric_Enums
parmenidianEnumerations
VERTEX_IN_DEGREE: Metric_Enums
VERTEX_OUT_DEGREE: Metric_Enums
VERTEX_DEGREE: Metric_Enums
VERTEX_BETWEENNESS: Metric_Enums
CLUSTERING_COEFFICIENT: Metric_Enums
EDGE_BETWEENNESS: Metric_Enums
GRAPH_DIAMETER: Metric_Enums
NUMBER_OF_VERTICES: Metric_Enums
NUMBER_OF_EDGES: Metric_Enums
NUMBER_OF_CONNECTED_COMPONENTS: Metric_Enums
NUMBER_OF_VERTICES_IN_GCC: Metric_Enums
NUMBER_OF_EDGES_IN_GCC: Metric_Enums
Metric_Enums()
<<Java Enumeration>>
Status
parmenidianEnumerations
UNDEFINED: Status
CREATION: Status
DELETION: Status
UPDATE: Status
value: int
Status(int)
getValue():int

Figure 3.8 Class Diagram of the ParmenidianEnumerations Package

3.2.3 Classes Collaborations Responsibilities (CRC) Method

A more fine – grained analysis that is not constrained within the limits of a package is carried out to provide a more comprehensive picture of the relationships between the elements of the different subsystems. Thus, the *CRC* method [BeCu89] is applied for each class of each package of Parmenidian Truth tool. In the context of the refactoring process, *CRC* cards are expected to be useful in our attempt to acquire a general overview of the responsibilities assigned to each class and also a more clear perception of the object interactions. In this way, we seek to identify classes which might encompass more responsibilities than these that are supposed to discharge and evaluate the degree of the coupling among objects.

Figures 3.9, 3.10, 3.11, 3.12, 3.13 and 3.14 depict the *CRC* cards for each package of the tool.

ExportManager	
• manages objects responsible for exports	• export.HecateScript • export.PowerPointGenerator • export.VideoGenerator

ModelManager	
• manages objects, that contain graph information • loads existing graph • visualizes the graph • saves layout changes made by the user • saves reports with metrics	• model.DiachronicGraph

ParmenidianTruthManager	
• manages export and model packages' objects • makes the calls for every operation, that the gui offers	• core.ModelManager • core.ExportManager • parmenidianEnumerations.Metric_Enums

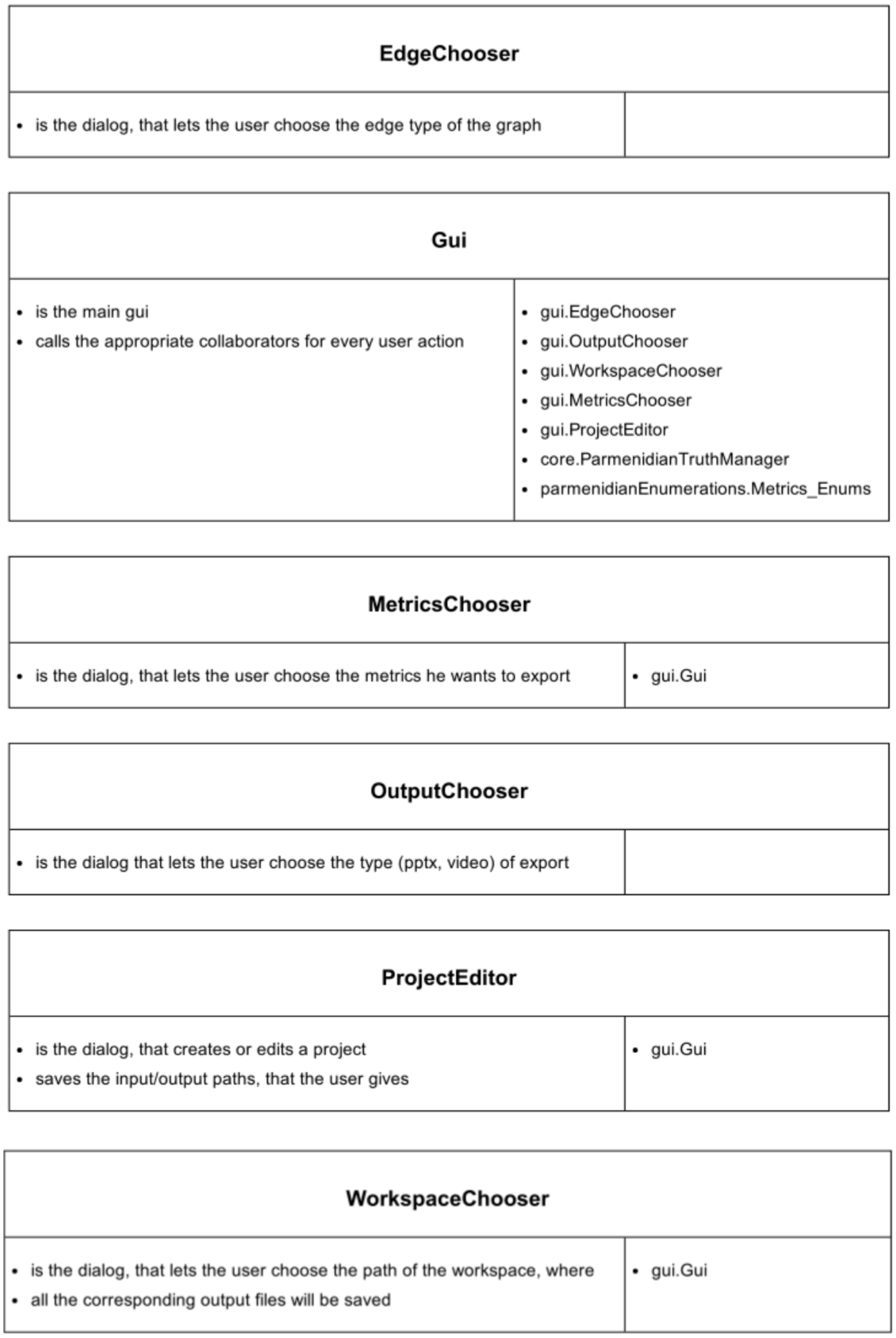
Figure 3.9 CRC Cards of the Classes of Package Core

HecateScript	
• creates the object, responsible for the transitions.xml file • makes the calls needed for Hecate • filters files	• model.Loader.HecateManager

PowerPointGenerator	
• creates powerpoint from screenshots • adds all the related features to the powerpoint	

VideoGenerator	
• extracts png files from the pptx file • creates the video from the png files • exports the video	

Figure 3.10 CRC Cards of the Classes of Package Export



EdgeChooser	
• is the dialog, that lets the user choose the edge type of the graph	

Gui	
• is the main gui • calls the appropriate collaborators for every user action	• gui.EdgeChooser • gui.OutputChooser • gui.WorkspaceChooser • gui.MetricsChooser • gui.ProjectEditor • core.ParmenidianTruthManager • parmenidianEnumerations.Metrics_Enums

MetricsChooser	
• is the dialog, that lets the user choose the metrics he wants to export	• gui.Gui

OutputChooser	
• is the dialog that lets the user choose the type (pptx, video) of export	

ProjectEditor	
• is the dialog, that creates or edits a project • saves the input/output paths, that the user gives	• gui.Gui

WorkspaceChooser	
• is the dialog, that lets the user choose the path of the workspace, where • all the corresponding output files will be saved	• gui.Gui

Figure 3.11 CRC Cards of the Classes of Package Gui

GraphmlLoader	
• loads the graphml file, that contains the coordinates • defines the coordinates of each table(node)	• model.Table • model.ForeignKey

HecateManager	
• makes Hecate calls • parses sql and xml files • produces the transitions and the evolution information of the schema via Hecate	• model.DBVersion • model.ForeignKey • model.Table • externalTools.HecateParser • externalTools.Schema • externalTools.Delta • externalTools.TransitionList • externalTools.Transitions • externalTools.Deletion • externalTools.DiffResult • externalTools.Transition • externalTools.Insersion • externalTools.Update • externalTools.ForeignKey • externalTools.Table • parmenidianEnumerations.Status

Parser	
• parses Hecate's output files • stores the information of the aforementioned files in model package's objects	• model.DBVersion • model.Loader.GraphmlLoader • model.Loader.HecateManager

Figure 3.12 CRC Cards of the Classes of Package Model.Loader

DBVersion	
• stores all the information, that a certain schema version has	• model.DBVersionVisualRepresentation
• produces all the metrics via the GraphMetrics object	• model.GraphMetrics
	• model.Table
	• model.ForeignKey
	• model.DiachronicGraph

DBVersionVisualRepresentation	
• visualizes the database schema as a graph	• model.Table
• creates png files for every version of the schema	• model.ForeignKey
	• model.DBVersion

DiachronicGraph	
• creates the diachronic graph	• model.DBVersion
• creates the reports with all the metrics	• model.Table
• manipulates the graph	• model.ForeignKey
	• model.DiachronicGraphVisualRepresentation
	• model.GraphMetrics
	• model.Loader.Parser
	• model.Loader.GraphmlLoader
	• parmenidianEnumerations.Status

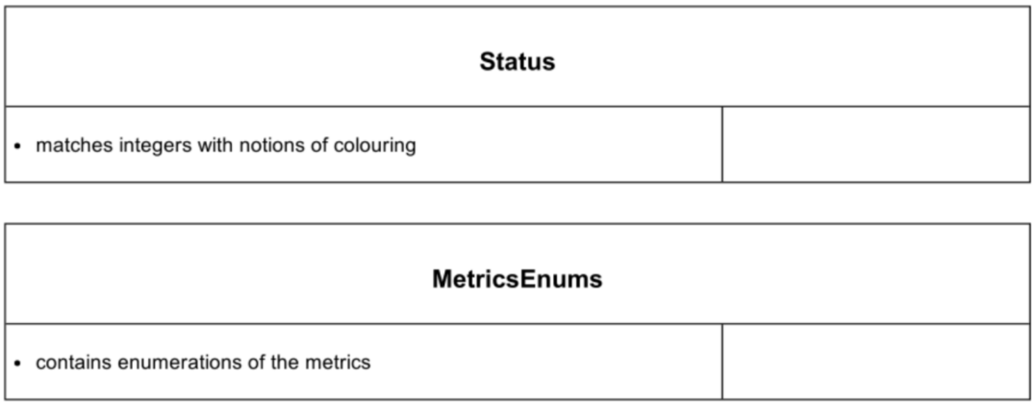
DiachronicGraphVisualRepresentation	
• is the graphical representation of the diachronic graph	• model.Table
• manages the layout changes	• model.ForeignKey
• transforms the graph to png files	• model.DiachronicGraph

ForeignKey	
• holds the information of every foreign key	

GraphMetrics	
• calculates all the metrics based on the graph	• model.Table
	• model.ForeignKey

Table	
• holds the information of the table (name, coordinates etc.)	• parmenidianEnumerations.Status

Figure 3.13 CRC Cards of the Classes of Package Model



Status	
• matches integers with notions of colouring	

MetricsEnums	
• contains enumerations of the metrics	

Figure 3.14 CRC Cards of the Classes of Package ParmenidianEnumerations

3.3 Refactoring Actions

The previous representations were helpful in our attempt to detect design defects and prioritize them based on their frequency of occurrence and the implications they create in case of modifying the source code. The classification of the defects is based on the taxonomy of [FBB+99] and is as follows:

- Package Level Issues
- God Classes
- Lack of APIs
- Duplicated Code
- Misplaced Methods
- Redundant Components
- Convention Violations

The following sections describe the previous defects in detail and present the refactoring techniques applied for improving tool's design and increasing its adaptability to imminent extension or maintenance actions.

3.3.1 Package Level Issues

The refactoring process starts from the highest level of abstraction, which is the package level. In this level, the most obvious and important violation is the cyclic dependency between *model.Loader* and *model* packages. One possible and straightforward approach to deal with this defect would be to merge the two packages, especially if we take into account the strong correlation between them. However, this option would significantly increase the complexity of the new package's design.

We finally decided that it would be more efficient to identify the elements of the two packages that cause the cycle and transfer them to a new package. Moreover, the use of *model.Loader's* elements in the *DiachronicGraph* class of the *model* package created the aforementioned cyclic dependency which was broken through creating the *dataImport* package containing the classes of *model.Loader* and removing the dependencies of the *model's* classes from the new package in a higher level.

Another design weakness we observed was the total absence of cohesion between the classes of the *export* package. In order to increase the coherence of the package, we transferred the *ExportManager* class from the *core* package to the *export* one, based on the fact that this class is the common client of *export's* elements. It is noteworthy to mention that there appears to be no resemblance in the implementation of the *export* package's classes and as a result there were no any other available options to increase the cohesion of this package.

3.3.2 God Classes

The term "God class" refers to a class that encapsulates more than one responsibility, violating the *Single – Responsibility Principle* (SRP). In [MaMa06], the *Single – Responsibility Principle* is defined as "A class should have only one reason to change". According to this principle, each responsibility assigned to a class is considered to be a reason to modify the corresponding class. Every change in the requirements of a system is applied via altering the responsibilities of its modules, and if a module undertakes two or more responsibilities it would be difficult to adjust any kind of changes related to one responsibility in a way that would not affect parts of the module that fulfill other purposes.

The ordinary way of dealing with this kind of design defect is to discriminate the methods within a class that were created to serve different purposes and extract each group of methods in new classes.

In the initial version of the Parmenidian Truth system, a module that meets the criteria in order to be classified as a "God class" is that of *DiachronicGraph* in the *model* package. This class encapsulates responsibilities related to graph manipulation and also those for the generation of reports that include various graph metrics.

Figure 3.15 verifies this assertion by depicting methods and attributes of the *DiachronicGraph* class as squares and circles respectively, where each edge between a square and a circle denote that the method has access to the corresponding attribute.

It is obvious that there are two discrete clusters of methods, which do not have access in common attributes. This definitely shows that the methods of each cluster fulfill different requirements and an extraction of one of the two clusters in a new class is necessary. Our decision was to extract the group of methods related to the generation of reports, which consisted of two discrete sub-groups and contained additional defects that we describe in next subsections. As a result, the *DiachronicGraph* class remained only with graph – related responsibilities, increasing in this way the cohesion among the methods of the class and abiding by the SRP.

3.3.3 Lack of APIs

Another design defect we observed was the lack of APIs, whose presence in a system is considered to be crucial, especially when the requirements of the system have to be modified or expanded. APIs' main role is that of determining a set of functionalities that another class, called "client", needs and imposing the implementation of these methods in classes that implement them. In this way, it is feasible to make "client" classes independent from changes occurring in concrete classes and agnostic to the details of the implementation.

In a first step, we introduced an interface that serves as a contract between the *Gui* class of package *gui* and the *ParmenidianTruthManager* class of package *core*. The *IParmenidianTruth* interface contains methods required by *Gui* and implemented by *ParmenidianTruthManager*. In this way, the *Gui* class does not depend directly on classes of the *core* package and becomes independent of the changes made in these modules. In Figure 3.16 the class diagram of the previous classes after the insertion of the interface is shown.

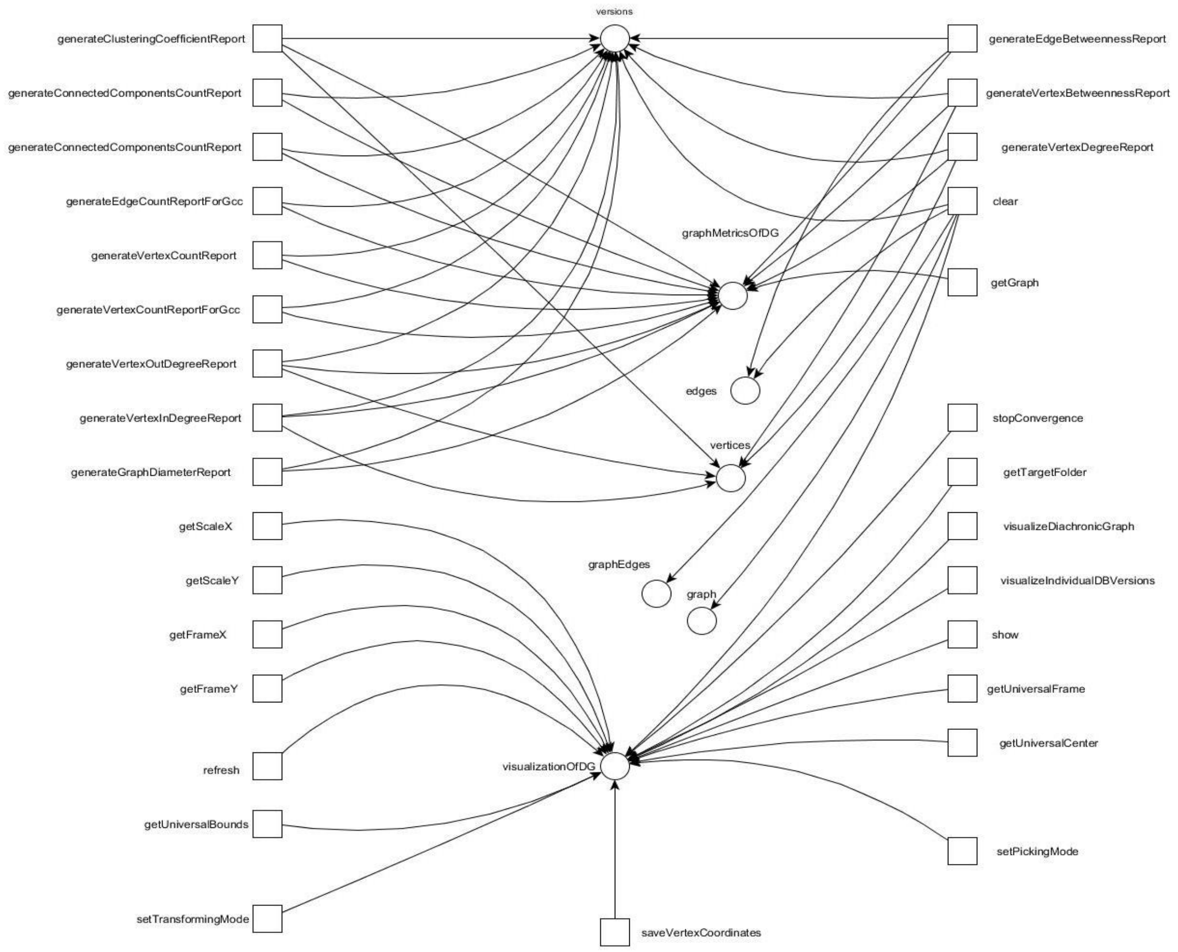
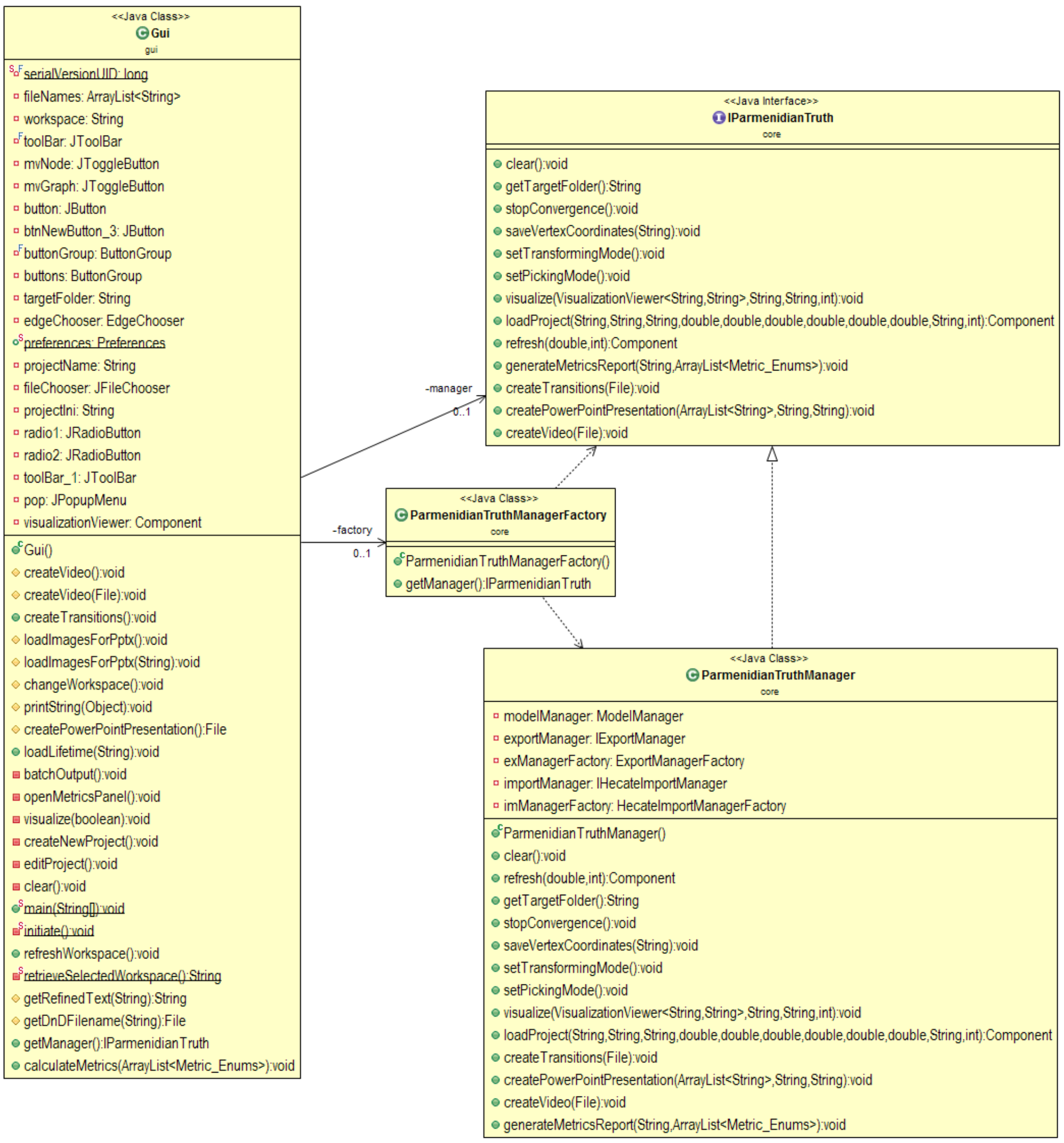

versions
generateClusteringCoefficientReport
generateConnectedComponentsCountReport
generateConnectedComponentsCountReport
generateEdgeCountReportForGcc
generateVertexCountReport
generateVertexCountReportForGcc
generateVertexOutDegreeReport
generateVertexInDegreeReport
generateGraphDiameterReport
getScaleX
getScaleY
getFrameX
getFrameY
refresh
getUniversalBounds
setTransformingMode
saveVertexCoordinates
visualizationOfDG
graphMetricsOfDG
edges
vertices
graphEdges
graph
generateEdgeBetweennessReport
generateVertexBetweennessReport
generateVertexDegreeReport
clear
getGraph
stopConvergence
getTargetFolder
visualizeDiachronicGraph
visualizeIndividualDBVersions
show
getUniversalFrame
getUniversalCenter
setPickingMode

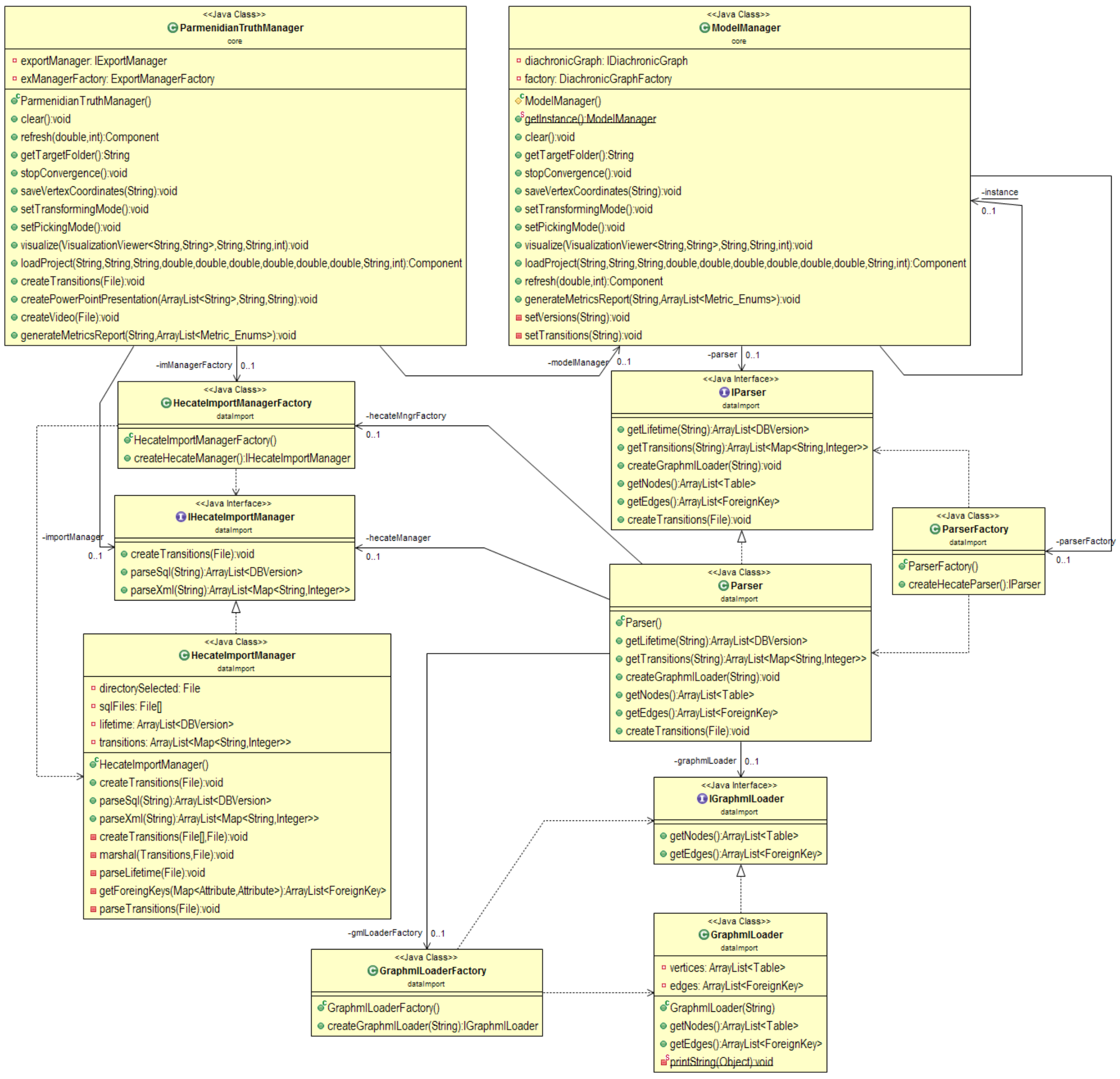
Figure 3.15 Methods (squares) and Attributes (circles) of the DiachronicGraph Class



<<Java Class>>
Gui
gui
serialVersionUID: long
fileNames: ArrayList<String>
workspace: String
toolBar: JToolBar
mvNode: JToggleButton
mvGraph: JToggleButton
button: JButton
btnNewButton_3: JButton
buttonGroup: ButtonGroup
buttons: ButtonGroup
targetFolder: String
edgeChooser: EdgeChooser
preferences: Preferences
projectName: String
fileChooser: JFileChooser
projectIni: String
radio1: JRadioButton
radio2: JRadioButton
toolBar_1: JToolBar
pop: JPopupMenu
visualizationViewer: Component
Gui()
createVideo():void
createVideo(File):void
createTransitions():void
loadImagesForPptx():void
loadImagesForPptx(String):void
changeWorkspace():void
printString(Object):void
createPowerPointPresentation():File
loadLifetime(String):void
batchOutput():void
openMetricsPanel():void
visualize(boolean):void
createNewProject():void
editProject():void
clear():void
main(String[]):void
initiate():void
refreshWorkspace():void
retrieveSelectedWorkspace():String
getRefinedText(String):String
getDnDFilename(String):File
getManager():IParmenidianTruth
calculateMetrics(ArrayList<Metric_Enums>):void
-manager
0..1
-factory
0..1
<<Java Interface>>
IParmenidianTruth
core
clear():void
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
refresh(double,int):Component
generateMetricsReport(String,ArrayList<Metric_Enums>):void
createTransitions(File):void
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
<<Java Class>>
ParmenidianTruthManagerFactory
core
ParmenidianTruthManagerFactory()
getManager():IParmenidianTruth
<<Java Class>>
ParmenidianTruthManager
core
modelManager: ModelManager
exportManager: IExportManager
exManagerFactory: ExportManagerFactory
importManager: IHecateImportManager
imManagerFactory: HecateImportManagerFactory
ParmenidianTruthManager()
clear():void
refresh(double,int):Component
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
createTransitions(File):void
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
generateMetricsReport(String,ArrayList<Metric_Enums>):void

Figure 3.16 Class Diagram of the IParmenidianTruth interface and its client

Our next change concerns the *dataImport* package, which consists of classes responsible for parsing sql files in order to create objects of the *model* package. The "clients" of this package are the *ModelManager* and the *ParmenidianTruthManager* of the *core* package and for this reason we created the interfaces *IParser* and *IHecateImportManager* implemented by the *Parser* and *HecateImportManager* classes respectively. The presence of the two interfaces is regarded necessary since the clients are different and we attempt to comply with the *Interface – Segregation Principle* (ISP), which, according to [Mart00], can be expressed as "Many client specific interfaces are better than one general purpose interface". The obedience of this principle ensures that we can avoid forming clients' dependencies upon methods that clients do not use and increase the cohesion within each interface. Apart from the two aforementioned interfaces, we added the *IGraphmlLoader* implemented by the *GraphmlLoader* class. In Figure 3.17, the class diagram of these interfaces along with their clients is depicted.



<<Java Class>>
ParmenidianTruthManager
core
exportManager: IExportManager
exManagerFactory: ExportManagerFactory
ParmenidianTruthManager()
clear():void
refresh(double,int):Component
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
createTransitions(File):void
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
generateMetricsReport(String,ArrayList<Metric_Enums>):void
<<Java Class>>
ModelManager
core
diachronicGraph: IDiachronicGraph
factory: DiachronicGraphFactory
ModelManager()
getInstance():ModelManager
clear():void
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
refresh(double,int):Component
generateMetricsReport(String,ArrayList<Metric_Enums>):void
setVersions(String):void
setTransitions(String):void
-instance
0..1
-imManagerFactory
0..1
-modelManager
0..1
-parser
0..1
<<Java Class>>
HecateImportManagerFactory
dataImport
HecateImportManagerFactory()
createHecateManager():IHecateImportManager
-hecateMngrFactory
0..1
<<Java Interface>>
IParser
dataImport
getLifetime(String):ArrayList<DBVersion>
getTransitions(String):ArrayList<Map<String,Integer>>
createGraphmlLoader(String):void
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
createTransitions(File):void
<<Java Interface>>
IHecateImportManager
dataImport
createTransitions(File):void
parseSql(String):ArrayList<DBVersion>
parseXml(String):ArrayList<Map<String,Integer>>
-importManager
0..1
-hecateManager
0..1
<<Java Class>>
ParserFactory
dataImport
ParserFactory()
createHecateParser():IParser
-parserFactory
0..1
<<Java Class>>
Parser
dataImport
Parser()
getLifetime(String):ArrayList<DBVersion>
getTransitions(String):ArrayList<Map<String,Integer>>
createGraphmlLoader(String):void
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
createTransitions(File):void
<<Java Class>>
HecateImportManager
dataImport
directorySelected: File
sqlFiles: File[]
lifetime: ArrayList<DBVersion>
transitions: ArrayList<Map<String,Integer>>
HecateImportManager()
createTransitions(File):void
parseSql(String):ArrayList<DBVersion>
parseXml(String):ArrayList<Map<String,Integer>>
createTransitions(File[],File):void
marshal(Transitions,File):void
parseLifetime(File):void
getForeingKeys(Map<Attribute,Attribute>):ArrayList<ForeignKey>
parseTransitions(File):void
-graphmlLoader
0..1
<<Java Interface>>
IGraphmlLoader
dataImport
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
-gmlLoaderFactory
0..1
<<Java Class>>
GraphmlLoader
dataImport
vertices: ArrayList<Table>
edges: ArrayList<ForeignKey>
GraphmlLoader(String)
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
printString(Object):void
<<Java Class>>
GraphmlLoaderFactory
dataImport
GraphmlLoaderFactory()
createGraphmlLoader(String):IGraphmlLoader

Figure 3.17 Class Diagram of the Interfaces of Package DataImport and their Clients

The *model* package is considered as the most complex package in terms of the dependencies between its classes, so it is crucial to recognize those modules that are important for the other packages and create interfaces that will determine the functionalities required by the clients and diminish the impact of possible changes introduced in classes of this package. Figures 3.18, 3.19 and 3.20 show the interfaces included in the *model* package.

Although *IDiachronicGraph* and *IMetricsReport* are used by the same client the functionalities they provide are uncorrelated between each other and that was the main reason for creating two interfaces instead of a large one that would be more prone to changes and less coherent.

The rationale for creating the *IGraphMetrics* interface was the fact that the existing implementations concerning the metrics produced by Parmenidian Truth tool were explicitly specified for csv files. In order to provide a set of methods that can be utilized in a subsequent different implementation, we created the *IGraphMetrics* interface with the role of clients assigned to the *DiachronicGraph* and the *DBVersion* classes.

The final introduction of an interface concerns the *export* package which contains classes responsible for creating a PowerPoint presentation and a video stream of the schema evolution of a database. The absence of cohesion between these classes was the main reason for transferring the *ExportManager* class from the *core* package to the *export* one and creating an interface that offers functionalities required by the *ParmenidianTruthManager* class. Figure 3.21 depicts the aforementioned interface with its client.

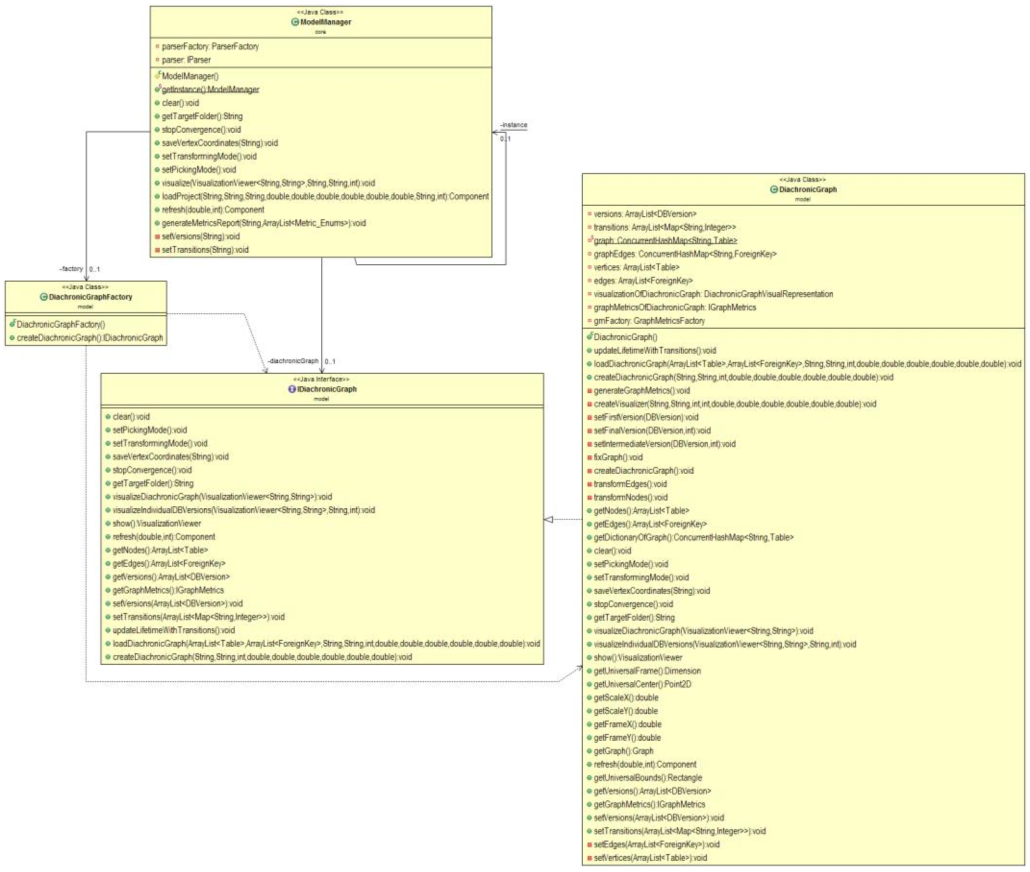

<<Java Class>>
ModelManager
core
parserFactory: ParserFactory
parser: IParser
ModelManager()
getInstance():ModelManager
clear():void
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
refresh(double,int):Component
generateMetricsReport(String,ArrayList<Metric_Enums>):void
setVersions(String):void
setTransitions(String):void
-instance
0..1
-factory
0..1
<<Java Class>>
DiachronicGraphFactory
model
DiachronicGraphFactory()
createDiachronicGraph():IDiachronicGraph
-diachronicGraph
0..1
<<Java Interface>>
IDiachronicGraph
model
clear():void
setPickingMode():void
setTransformingMode():void
saveVertexCoordinates(String):void
stopConvergence():void
getTargetFolder():String
visualizeDiachronicGraph(VisualizationViewer<String,String>):void
visualizeIndividualDBVersions(VisualizationViewer<String,String>,String,int):void
show():VisualizationViewer
refresh(double,int):Component
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
getVersions():ArrayList<DBVersion>
getGraphMetrics():IGraphMetrics
setVersions(ArrayList<DBVersion>):void
setTransitions(ArrayList<Map<String,Integer>>):void
updateLifetimeWithTransitions():void
loadDiachronicGraph(ArrayList<Table>,ArrayList<ForeignKey>,String,String,int,double,double,double,double,double,double):void
createDiachronicGraph(String,String,int,double,double,double,double,double,double):void
<<Java Class>>
DiachronicGraph
model
versions: ArrayList<DBVersion>
transitions: ArrayList<Map<String,Integer>>
graph: ConcurrentHashMap<String,Table>
graphEdges: ConcurrentHashMap<String,ForeignKey>
vertices: ArrayList<Table>
edges: ArrayList<ForeignKey>
visualizationOfDiachronicGraph: DiachronicGraphVisualRepresentation
graphMetricsOfDiachronicGraph: IGraphMetrics
gmFactory: GraphMetricsFactory
DiachronicGraph()
updateLifetimeWithTransitions():void
loadDiachronicGraph(ArrayList<Table>,ArrayList<ForeignKey>,String,String,int,double,double,double,double,double,double):void
createDiachronicGraph(String,String,int,double,double,double,double,double,double):void
generateGraphMetrics():void
createVisualizer(String,String,int,int,double,double,double,double,double,double):void
setFirstVersion(DBVersion):void
setFinalVersion(DBVersion,int):void
setIntermediateVersion(DBVersion,int):void
fixGraph():void
createDiachronicGraph():void
transformEdges():void
transformNodes():void
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
getDictionaryOfGraph():ConcurrentHashMap<String,Table>
clear():void
setPickingMode():void
setTransformingMode():void
saveVertexCoordinates(String):void
stopConvergence():void
getTargetFolder():String
visualizeDiachronicGraph(VisualizationViewer<String,String>):void
visualizeIndividualDBVersions(VisualizationViewer<String,String>,String,int):void
show():VisualizationViewer
getUniversalFrame():Dimension
getUniversalCenter():Point2D
getScaleX():double
getScaleY():double
getFrameX():double
getFrameY():double
getGraph():Graph
refresh(double,int):Component
getUniversalBounds():Rectangle
getVersions():ArrayList<DBVersion>
getGraphMetrics():IGraphMetrics
setVersions(ArrayList<DBVersion>):void
setTransitions(ArrayList<Map<String,Integer>>):void
setEdges(ArrayList<ForeignKey>):void
setVertices(ArrayList<Table>):void

Figure 3.18 Class Diagram of the IDiachronicGraph Interface and its Client

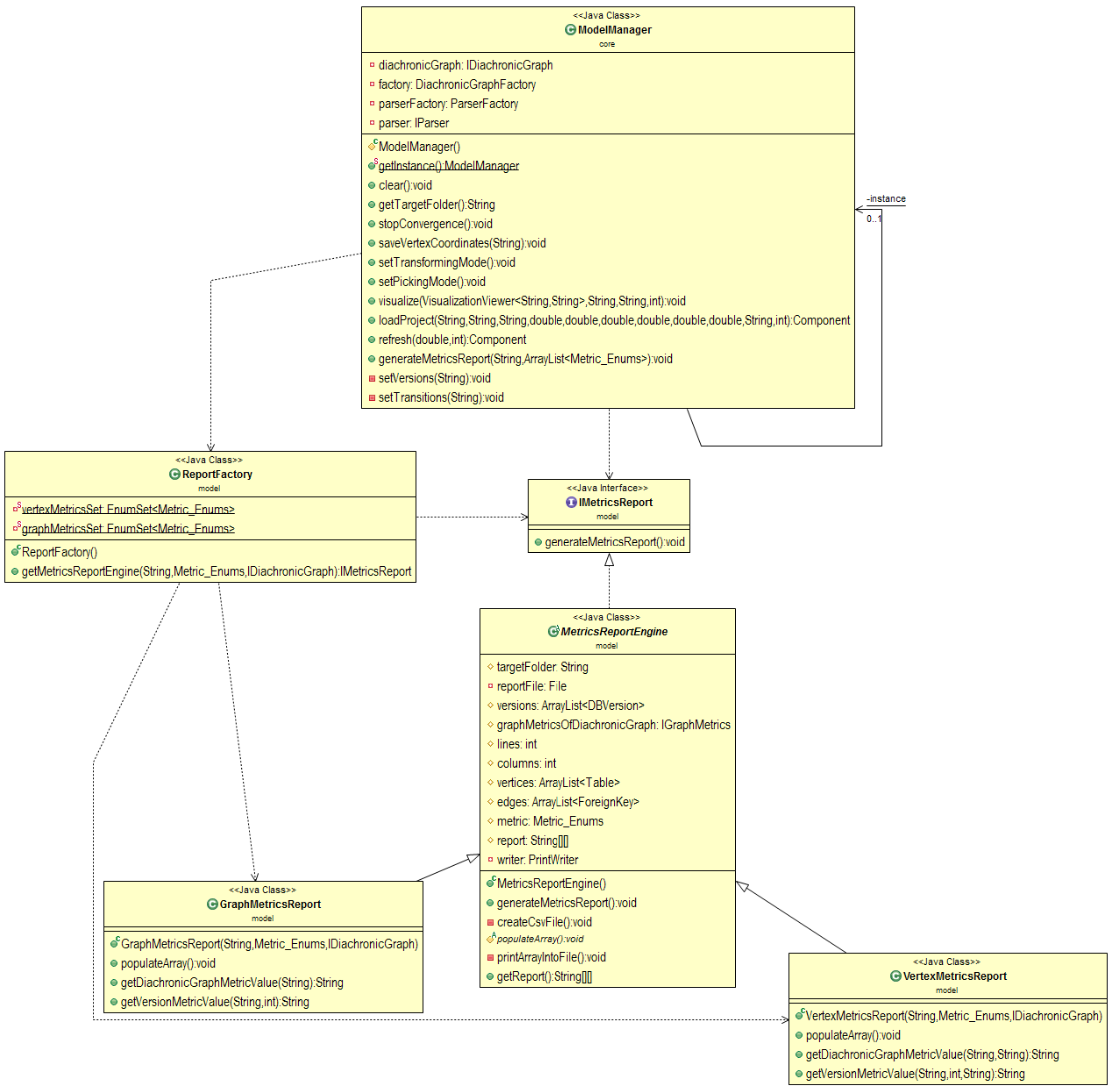
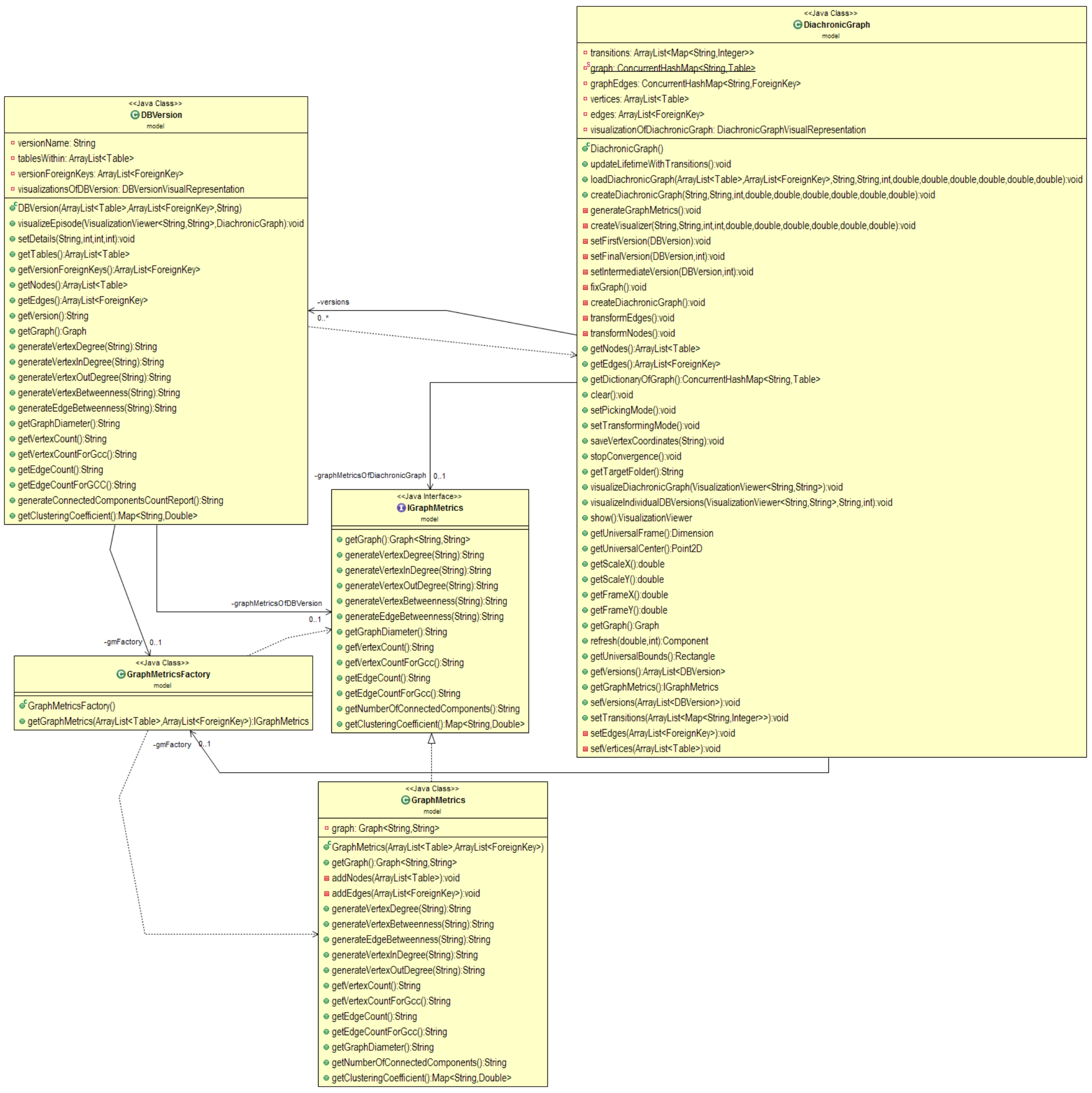
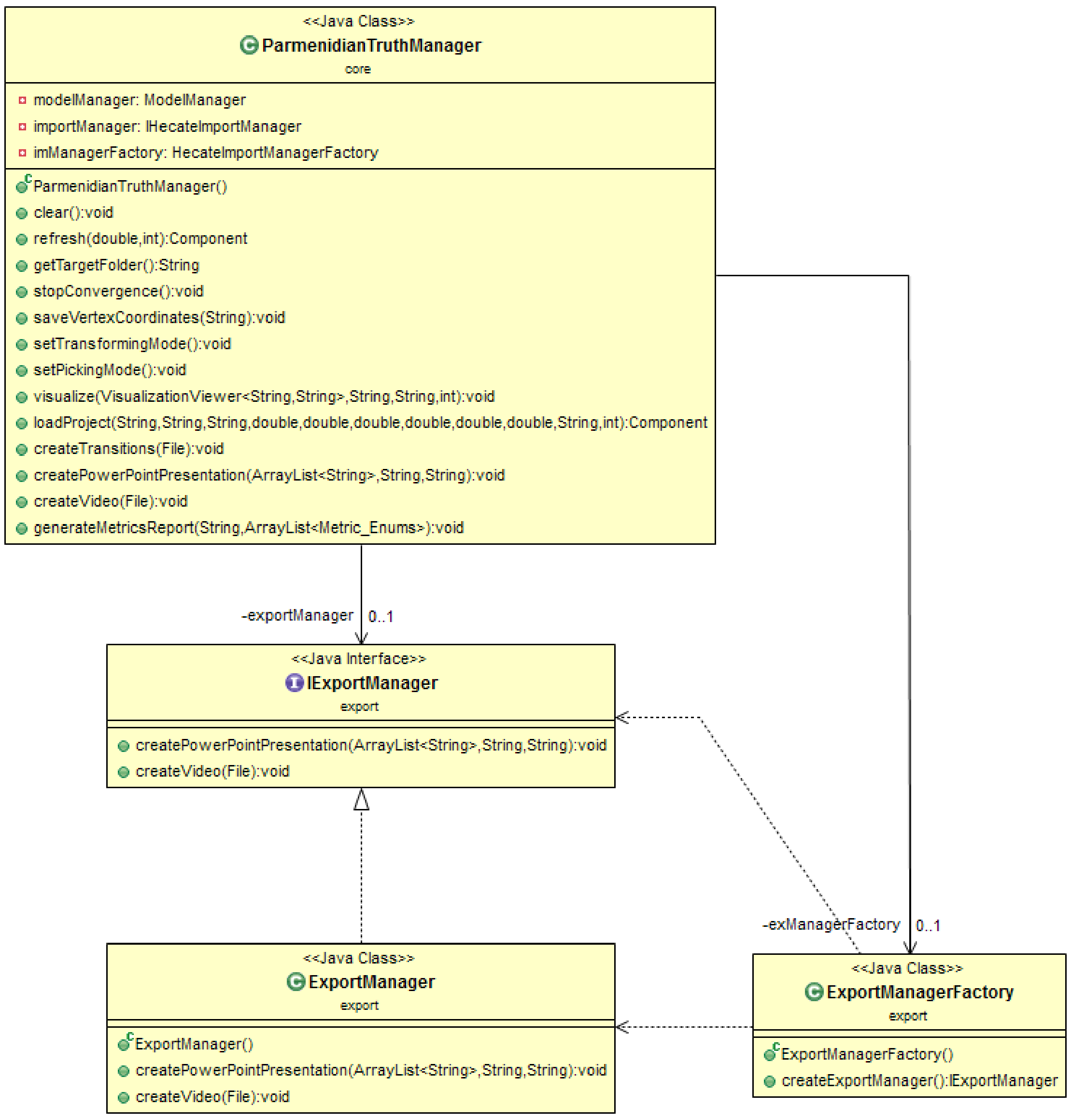

<<Java Class>>
ModelManager
core
diachronicGraph: IDiachronicGraph
factory: DiachronicGraphFactory
parserFactory: ParserFactory
parser: IParser
ModelManager()
getInstance():ModelManager
clear():void
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
refresh(double,int):Component
generateMetricsReport(String,ArrayList<Metric_Enums>):void
setVersions(String):void
setTransitions(String):void
-instance
0..1
<<Java Class>>
ReportFactory
model
vertexMetricsSet: EnumSet<Metric_Enums>
graphMetricsSet: EnumSet<Metric_Enums>
ReportFactory()
getMetricsReportEngine(String,Metric_Enums,IDiachronicGraph):IMetricsReport
<<Java Interface>>
IMetricsReport
model
generateMetricsReport():void
<<Java Class>>
MetricsReportEngine
model
targetFolder: String
reportFile: File
versions: ArrayList<DBVersion>
graphMetricsOfDiachronicGraph: IGraphMetrics
lines: int
columns: int
vertices: ArrayList<Table>
edges: ArrayList<ForeignKey>
metric: Metric_Enums
report: String[][]
writer: PrintWriter
MetricsReportEngine()
generateMetricsReport():void
createCsvFile():void
populateArray():void
printArrayIntoFile():void
getReport():String[][]
<<Java Class>>
GraphMetricsReport
model
GraphMetricsReport(String,Metric_Enums,IDiachronicGraph)
populateArray():void
getDiachronicGraphMetricValue(String):String
getVersionMetricValue(String,int):String
<<Java Class>>
VertexMetricsReport
model
VertexMetricsReport(String,Metric_Enums,IDiachronicGraph)
populateArray():void
getDiachronicGraphMetricValue(String,String):String
getVersionMetricValue(String,int,String):String

Figure 3.19 Class Diagram of the IMetricsReport Interface and its Client



<<Java Class>>
DiachronicGraph
model
transitions: ArrayList<Map<String,Integer>>
graph: ConcurrentHashMap<String,Table>
graphEdges: ConcurrentHashMap<String,ForeignKey>
vertices: ArrayList<Table>
edges: ArrayList<ForeignKey>
visualizationOfDiachronicGraph: DiachronicGraphVisualRepresentation
DiachronicGraph()
updateLifetimeWithTransitions():void
loadDiachronicGraph(ArrayList<Table>,ArrayList<ForeignKey>,String,String,int,double,double,double,double,double,double):void
createDiachronicGraph(String,String,int,double,double,double,double,double,double):void
generateGraphMetrics():void
createVisualizer(String,String,int,int,double,double,double,double,double,double):void
setFirstVersion(DBVersion):void
setFinalVersion(DBVersion,int):void
setIntermediateVersion(DBVersion,int):void
fixGraph():void
createDiachronicGraph():void
transformEdges():void
transformNodes():void
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
getDictionaryOfGraph():ConcurrentHashMap<String,Table>
clear():void
setPickingMode():void
setTransformingMode():void
saveVertexCoordinates(String):void
stopConvergence():void
getTargetFolder():String
visualizeDiachronicGraph(VisualizationViewer<String,String>):void
visualizeIndividualDBVersions(VisualizationViewer<String,String>,String,int):void
show():VisualizationViewer
getUniversalFrame():Dimension
getUniversalCenter():Point2D
getScaleX():double
getScaleY():double
getFrameX():double
getFrameY():double
getGraph():Graph
refresh(double,int):Component
getUniversalBounds():Rectangle
getVersions():ArrayList<DBVersion>
getGraphMetrics():IGraphMetrics
setVersions(ArrayList<DBVersion>):void
setTransitions(ArrayList<Map<String,Integer>>):void
setEdges(ArrayList<ForeignKey>):void
setVertices(ArrayList<Table>):void
<<Java Class>>
DBVersion
model
versionName: String
tablesWithin: ArrayList<Table>
versionForeignKeys: ArrayList<ForeignKey>
visualizationsOfDBVersion: DBVersionVisualRepresentation
DBVersion(ArrayList<Table>,ArrayList<ForeignKey>,String)
visualizeEpisode(VisualizationViewer<String,String>,DiachronicGraph):void
setDetails(String,int,int,int):void
getTables():ArrayList<Table>
getVersionForeignKeys():ArrayList<ForeignKey>
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
getVersion():String
getGraph():Graph
generateVertexDegree(String):String
generateVertexInDegree(String):String
generateVertexOutDegree(String):String
generateVertexBetweenness(String):String
generateEdgeBetweenness(String):String
getGraphDiameter():String
getVertexCount():String
getVertexCountForGcc():String
getEdgeCount():String
getEdgeCountForGCC():String
generateConnectedComponentsCountReport():String
getClusteringCoefficient():Map<String,Double>
-versions
0..*
-graphMetricsOfDiachronicGraph
0..1
<<Java Interface>>
IGraphMetrics
model
getGraph():Graph<String,String>
generateVertexDegree(String):String
generateVertexInDegree(String):String
generateVertexOutDegree(String):String
generateVertexBetweenness(String):String
generateEdgeBetweenness(String):String
getGraphDiameter():String
getVertexCount():String
getVertexCountForGcc():String
getEdgeCount():String
getEdgeCountForGcc():String
getNumberOfConnectedComponents():String
getClusteringCoefficient():Map<String,Double>
-graphMetricsOfDBVersion
0..1
-gmFactory
0..1
<<Java Class>>
GraphMetricsFactory
model
GraphMetricsFactory()
getGraphMetrics(ArrayList<Table>,ArrayList<ForeignKey>):IGraphMetrics
-gmFactory
0..1
<<Java Class>>
GraphMetrics
model
graph: Graph<String,String>
GraphMetrics(ArrayList<Table>,ArrayList<ForeignKey>)
getGraph():Graph<String,String>
addNodes(ArrayList<Table>):void
addEdges(ArrayList<ForeignKey>):void
generateVertexDegree(String):String
generateVertexBetweenness(String):String
generateEdgeBetweenness(String):String
generateVertexInDegree(String):String
generateVertexOutDegree(String):String
getVertexCount():String
getVertexCountForGcc():String
getEdgeCount():String
getEdgeCountForGcc():String
getGraphDiameter():String
getNumberOfConnectedComponents():String
getClusteringCoefficient():Map<String,Double>

Figure 3.20 Class Diagram of the IGraphMetrics Interface and its Clients



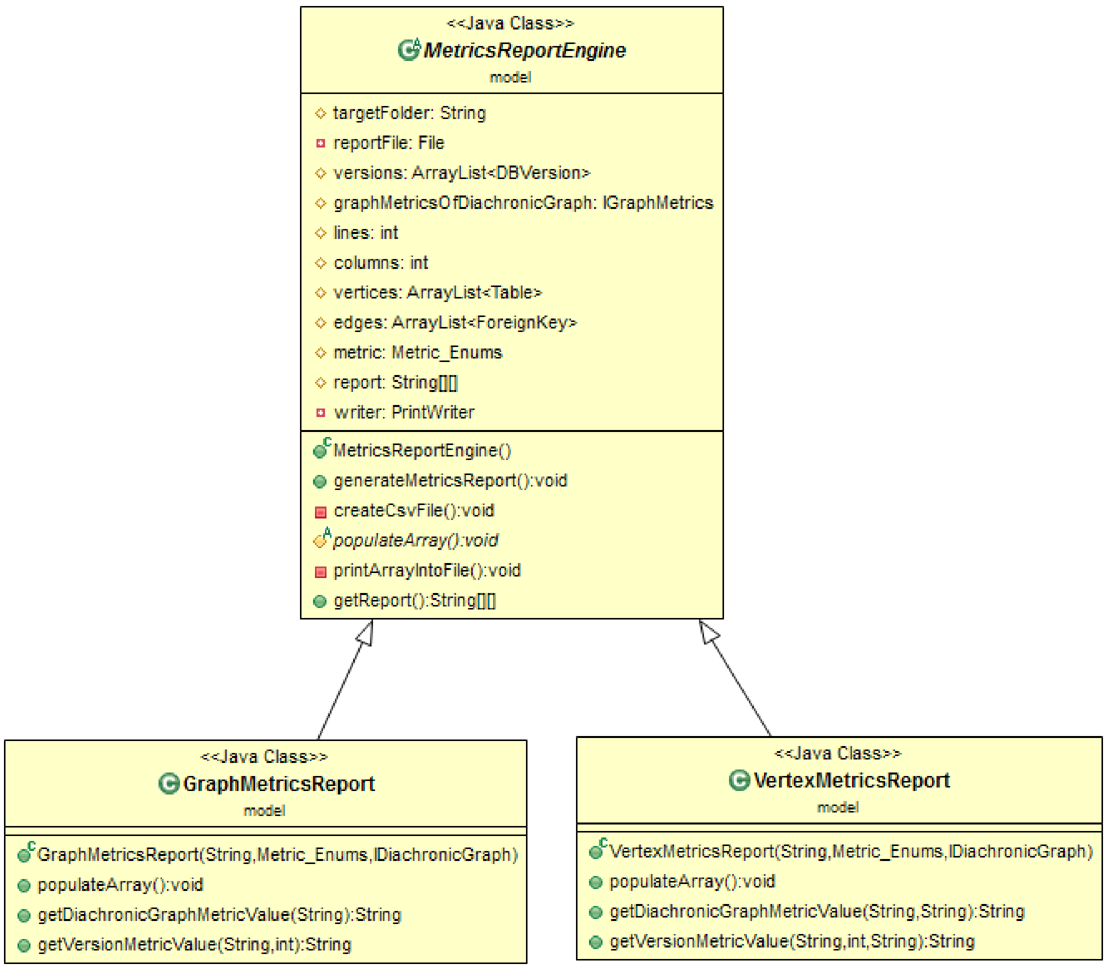
<<Java Class>>
ParmenidianTruthManager
core
modelManager: ModelManager
importManager: IHecateImportManager
imManagerFactory: HecateImportManagerFactory
ParmenidianTruthManager()
clear():void
refresh(double,int):Component
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
createTransitions(File):void
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
generateMetricsReport(String,ArrayList<Metric_Enums>):void
-exportManager
0..1
<<Java Interface>>
IExportManager
export
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
-exManagerFactory
0..1
<<Java Class>>
ExportManager
export
ExportManager()
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
<<Java Class>>
ExportManagerFactory
export
ExportManagerFactory()
createExportManager():IExportManager

Figure 3.21 Class Diagram of the IExportManager Interface and its client

3.3.4 Duplicated Code

The implementation of the methods which are responsible for the generation of the reports that contain various metrics consists of three discrete parts. The first one is related to the creation of the csv file which contains the results. The second part includes the computation of the metrics selected by the user. Finally, the third part registers the results of the second part to the file created in the first part. Irrespective of the metrics chosen, the first and the third parts are implemented in the same way for all the different metrics, while the second one can be classified in two categories (specifically (a) metrics concerning the entire graph and (b) metrics related to individual nodes) as far as its implementation is concerned. It is obvious that this part is an example of duplicated code.

Taking into account the recommended methods for dealing with duplicated code, we created the *MetricsReportEngine* abstract class. This class contains a template method, which defines the execution order of the aforementioned parts. As mentioned, the first and the third parts are the same for all the metrics and for this reason they are implemented in the abstract class. On the contrary, the second part for the metrics computation is separated into graph and vertex related implementations. This difference resulted in the creation of the subclasses *GraphMetricsReport* and *VertexMetricsReport*, each of them implementing the corresponding metric computation related code. Figure 3.22 shows the class diagram of the previous classes.



<<Java Class>>
MetricsReportEngine
model
targetFolder: String
reportFile: File
versions: ArrayList<DBVersion>
graphMetricsOfDiachronicGraph: IGraphMetrics
lines: int
columns: int
vertices: ArrayList<Table>
edges: ArrayList<ForeignKey>
metric: Metric_Enums
report: String[][]
writer: PrintWriter
MetricsReportEngine()
generateMetricsReport():void
createCsvFile():void
populateArray():void
printArrayIntoFile():void
getReport():String[][]
<<Java Class>>
GraphMetricsReport
model
GraphMetricsReport(String,Metric_Enums,IDiachronicGraph)
populateArray():void
getDiachronicGraphMetricValue(String):String
getVersionMetricValue(String,int):String
<<Java Class>>
VertexMetricsReport
model
VertexMetricsReport(String,Metric_Enums,IDiachronicGraph)
populateArray():void
getDiachronicGraphMetricValue(String,String):String
getVersionMetricValue(String,int,String):String

Figure 3.22 Class Diagram of the Classes Responsible for Metrics Reports Generation

Another occurrence of duplicated code was identified in the *HecateManager* and the *HecateScript* classes, which both contained the same auxiliary class, called *SQLFileFilter*. One of the packages included in ParmenidianTruth tool was the *fileFilter* package containing the *SQLFileFilter* and the *ImageFileFilter* classes used by the *VideoGenerator* class. However, this package remained unused and in order to eliminate the duplicated code, the classes of the *fileFilter* package can be utilized instead of the auxiliary ones.

3.3.5 Misplaced Methods

As described in subsection 3.3.2, the *DiachronicGraph* class consisted of methods responsible for the generation of metrics reports and methods used for graph manipulation functionalities. From our perspective, the methods related to the generation of reports resided in a class irrelevant to the functionality they offer and it would be more sensible to be assigned in new classes described in subsection 3.3.4.

3.3.6 Redundant Components

The *hecateImports* package existed in ParmenidianTruth's source code, containing all the classes provided by the *externalTools* package. The classes of the *hecateImports* package were not exploited by the other packages and for this reason we decided to remove it.

3.3.7 Convention Violations

As far as the conventions abidance is concerned, we utilized a checkstyle tool created by A. Papamichail, so as to identify potential violations that exist in ParmenidianTruth. These violations concern the following conventions:

- Name conventions
- Method parameter conventions
- Class size conventions

Figure 3.23 depicts the results provided by the tool prior and after the refactoring process. The horizontal axis includes the name of each class and the vertical the number of the violations detected.

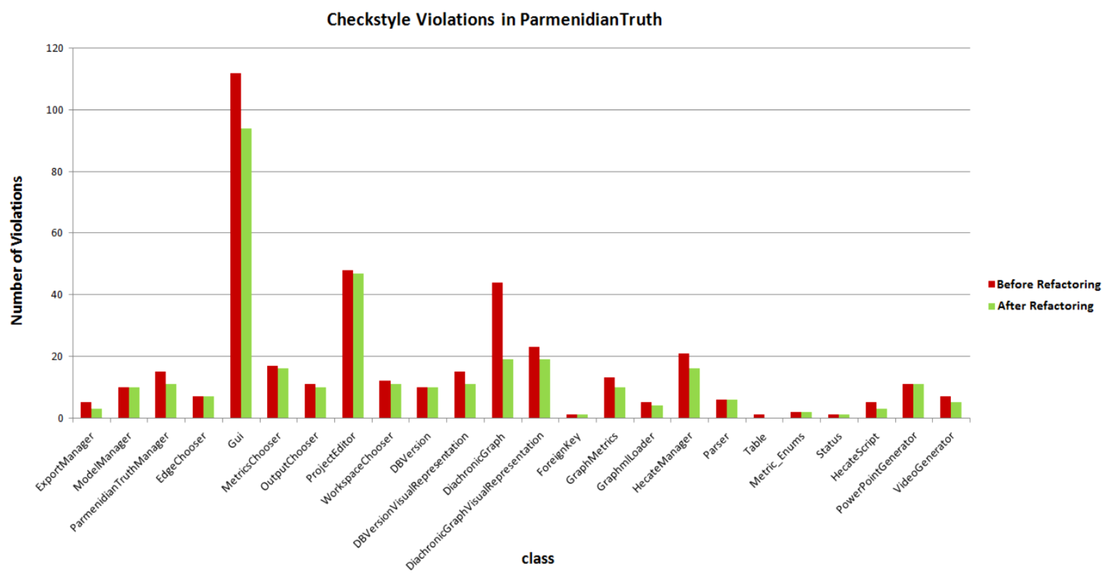


Figure 3.23 Checkstyle Violations Before and After the Refactoring Process

3.4 Testing

In this section, we describe the tests we applied in order to evaluate the correctness of our modifications. Using the unit testing framework for Java, *JUnit*, we created a test case for each of the classes that we had either added or changed. In most cases, we utilized *Mockito* [Fabe07], a mocking framework that allowed us to create objects that simulate the behavior of real objects, without their dependencies.

The *ReportFactory* class contains only one method that creates the object responsible for the generation of metrics reports. Using a mock object of the *DiachronicGraph* class, we confirmed that this object is created correctly.

As for the tests performed for the abstract class that determines the execution order for the creation of the reports, *MetricsReportEngine*, and its subclasses *GraphMetricsReport* and *VertexMetricsReport*, we used mocking as well as spying techniques. *Spying* is a functionality provided by the Mockito framework and allows us to call all the normal methods of an object while still tracking every interaction. The tests for the subclasses examined the initializations and the non – void methods. For testing the creation of objects that generate graph and vertex metrics reports we used spies that let us monitor the calculation of the metrics.

Apart from the tests designed for the new classes, we also assessed the behavior of the *DiachronicGraph* class which was the subject to our most alterations. The results of the testing process confirmed that the object construction and the operation of the methods involved are the expected ones.

Finally, except for the *JUnit* tests, we performed black – box testing for all the parts that we modified and were responsible for the creation of the metrics reports. More precisely, we compared the files that ParmenidianTruth exported prior to our modifications with the ones created after our modifications. In all cases, each of them concerning different dataset, there was no difference between the corresponding files.

3.5 Final Architecture and Design

3.5.1 Package Diagram

This section includes the final high – level architecture of ParmenidianTruth via the package diagram along with the corresponding dependencies, depicted in Figure 3.24.

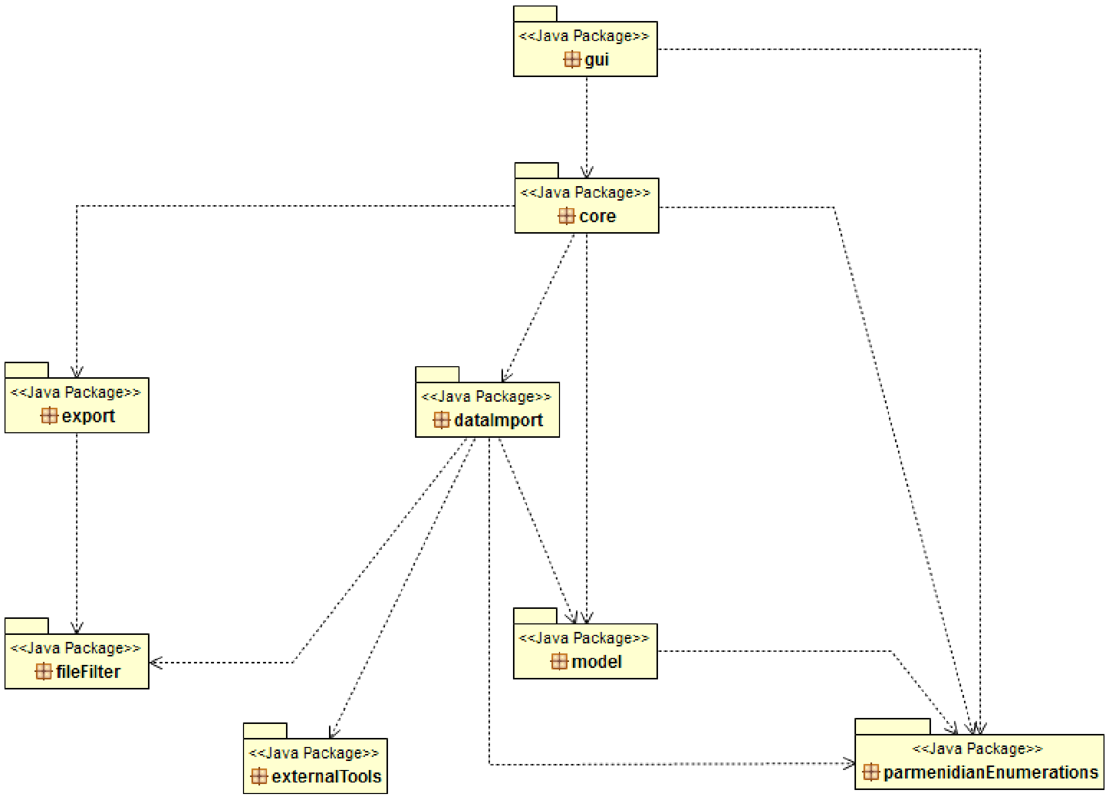


Figure 3.24 Updated Package Diagram of ParmenidianTruth

3.5.2 Class Diagrams

The following class diagrams present the new structure of each package of ParmenidianTruth. The *parmenidianEnumerations* and *externalTools* packages are omitted due to the fact that they were not altered during the refactoring process and so their internal structure remained identical to the previous one.

Figure 3.25 depicts the class diagram of the *gui* package.

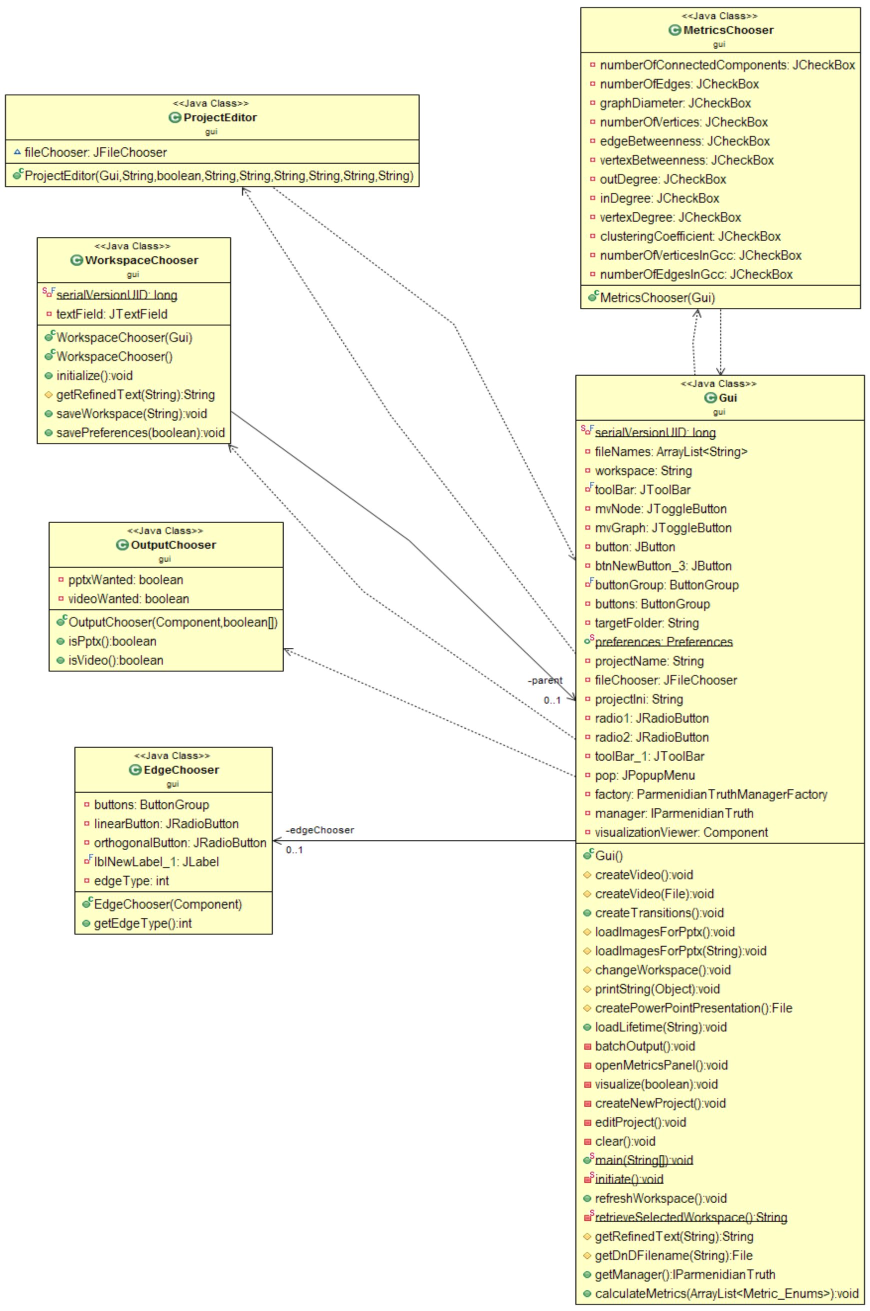
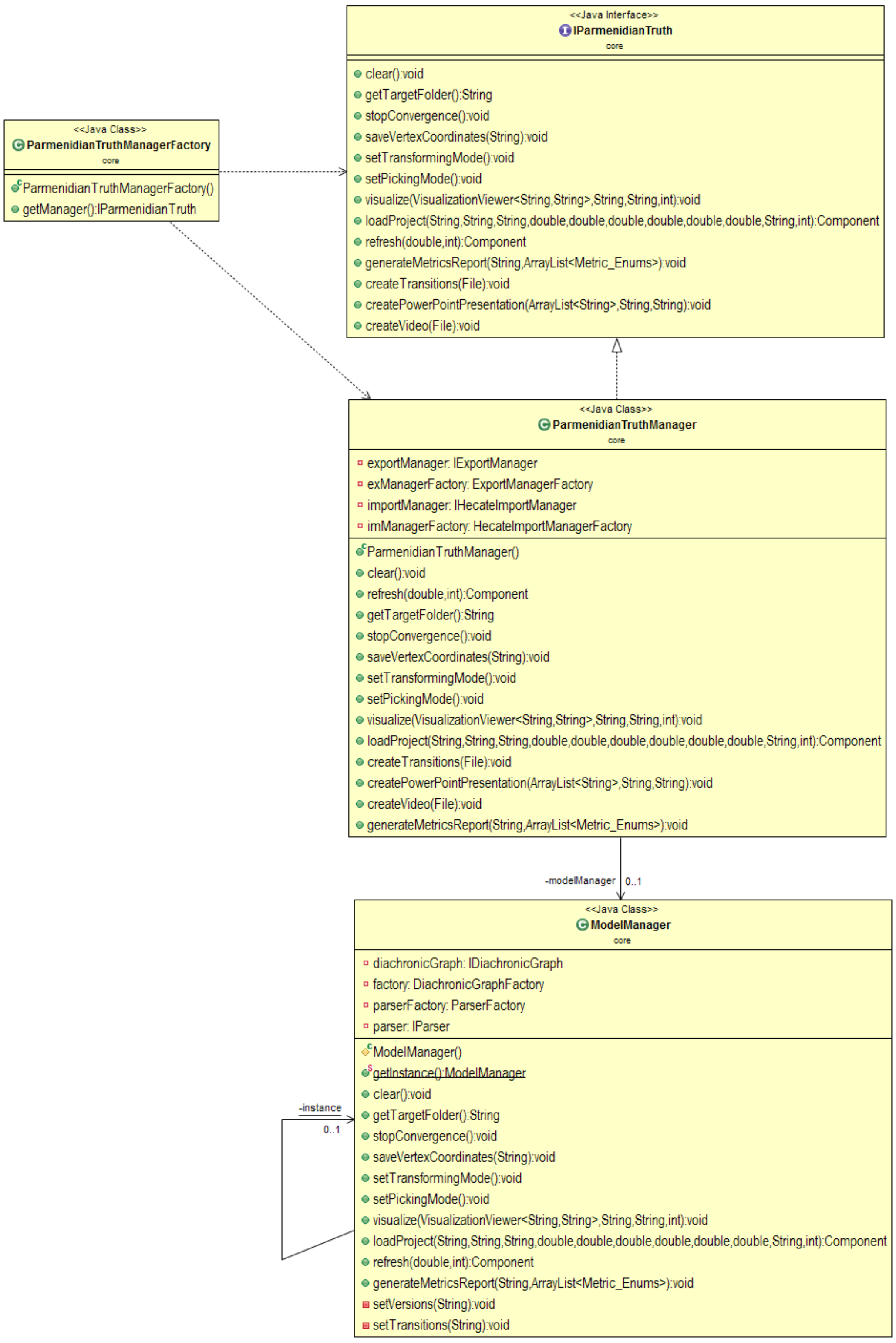


Figure 3.25 Updated Class Diagram of the Gui Package

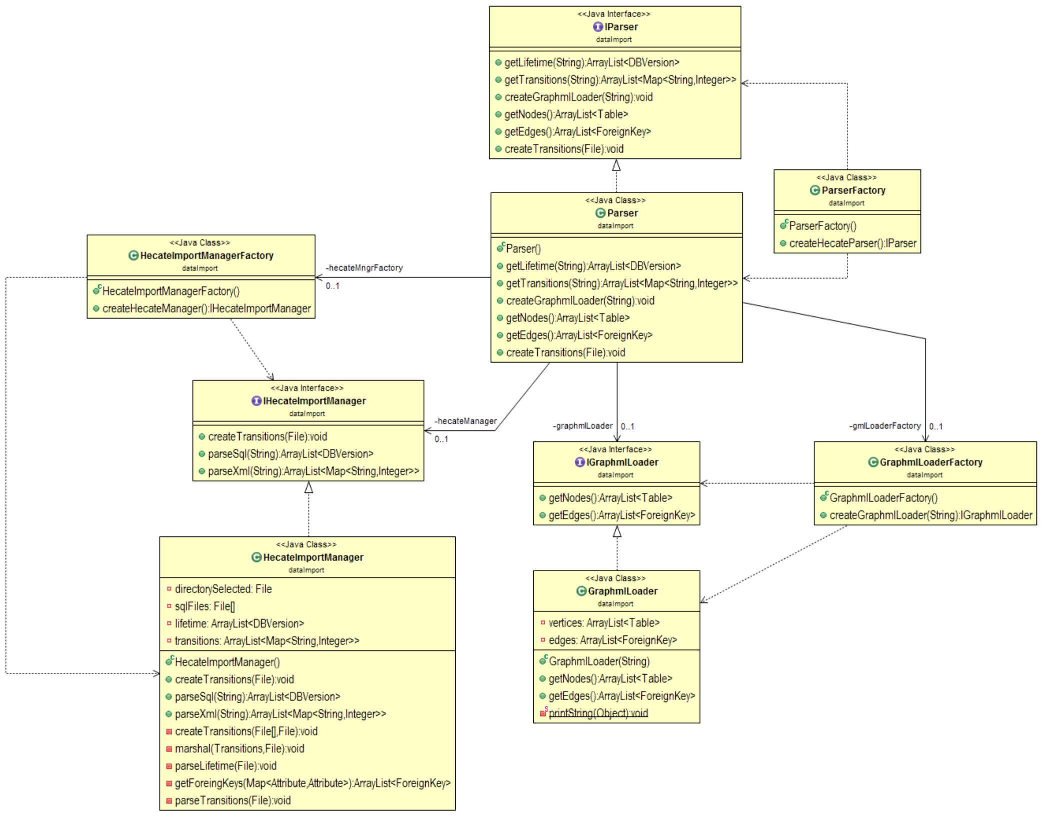
Figure 3.26 shows the class diagram of the *core* package.



<<Java Interface>>
IParmenidianTruth
core
clear():void
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
refresh(double,int):Component
generateMetricsReport(String,ArrayList<Metric_Enums>):void
createTransitions(File):void
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
<<Java Class>>
ParmenidianTruthManagerFactory
core
ParmenidianTruthManagerFactory()
getManager():IParmenidianTruth
<<Java Class>>
ParmenidianTruthManager
core
exportManager: IExportManager
exManagerFactory: ExportManagerFactory
importManager: IHecateImportManager
imManagerFactory: HecateImportManagerFactory
ParmenidianTruthManager()
clear():void
refresh(double,int):Component
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
createTransitions(File):void
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
generateMetricsReport(String,ArrayList<Metric_Enums>):void
-modelManager
0..1
<<Java Class>>
ModelManager
core
diachronicGraph: IDiachronicGraph
factory: DiachronicGraphFactory
parserFactory: ParserFactory
parser: IParser
ModelManager()
getInstance():ModelManager
clear():void
getTargetFolder():String
stopConvergence():void
saveVertexCoordinates(String):void
setTransformingMode():void
setPickingMode():void
visualize(VisualizationViewer<String,String>,String,String,int):void
loadProject(String,String,String,double,double,double,double,double,double,String,int):Component
refresh(double,int):Component
generateMetricsReport(String,ArrayList<Metric_Enums>):void
setVersions(String):void
setTransitions(String):void
-instance
0..1

Figure 3.26 Updated Class Diagram of the Core Package

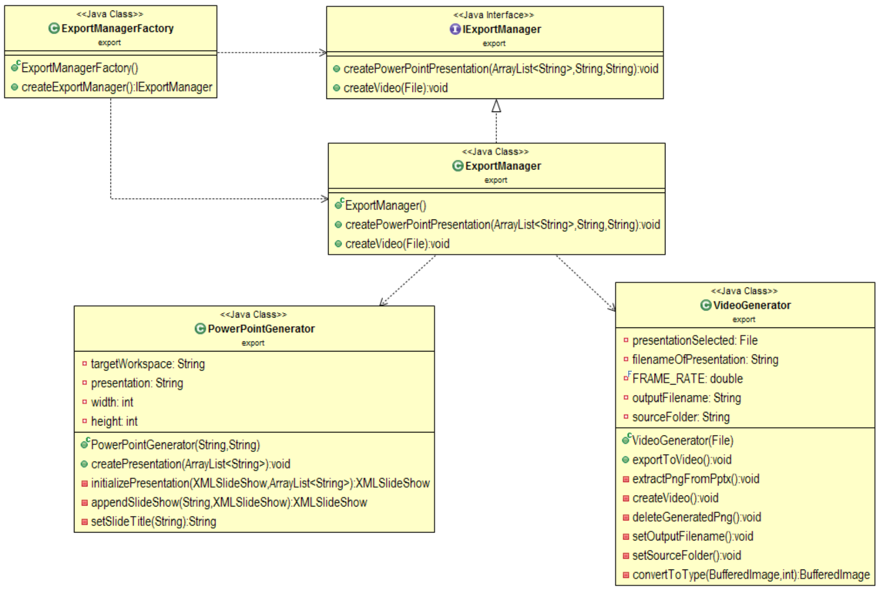
Figure 3.27 presents the class diagram of the *dataImport* package.



<<Java Interface>> IParser dataimport
getLifetime(String):ArrayList<DBVersion>
getTransitions(String):ArrayList<Map<String,Integer>>
createGraphmlLoader(String):void
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
createTransitions(File):void
<<Java Class>> ParserFactory dataimport
ParserFactory()
createHecateParser():IParser
<<Java Class>> Parser dataimport
Parser()
getLifetime(String):ArrayList<DBVersion>
getTransitions(String):ArrayList<Map<String,Integer>>
createGraphmlLoader(String):void
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
createTransitions(File):void
<<Java Class>> HecateImportManagerFactory dataimport
HecateImportManagerFactory()
createHecateManager():IHecateImportManager
-hecateMngrFactory 0..1
<<Java Interface>> IHecateImportManager dataimport
createTransitions(File):void
parseSql(String):ArrayList<DBVersion>
parseXml(String):ArrayList<Map<String,Integer>>
-hecateManager 0..1
-graphmlLoader 0..1
-gmlLoaderFactory 0..1
<<Java Interface>> IGraphmlLoader dataimport
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
<<Java Class>> GraphmlLoaderFactory dataimport
GraphmlLoaderFactory()
createGraphmlLoader(String):IGraphmlLoader
<<Java Class>> HecateImportManager dataimport
directorySelected: File
sqlFiles: File[]
lifetime: ArrayList<DBVersion>
transitions: ArrayList<Map<String,Integer>>
HecateImportManager()
createTransitions(File):void
parseSql(String):ArrayList<DBVersion>
parseXml(String):ArrayList<Map<String,Integer>>
createTransitions(File[],File):void
marshal(Transitions,File):void
parseLifetime(File):void
getForeingKeys(Map<Attribute,Attribute>):ArrayList<ForeignKey>
parseTransitions(File):void
<<Java Class>> GraphmlLoader dataimport
vertices: ArrayList<Table>
edges: ArrayList<ForeignKey>
GraphmlLoader(String)
getNodes():ArrayList<Table>
getEdges():ArrayList<ForeignKey>
printString(Object):void

Figure 3.27 Updated Class Diagram of the DataImport Package

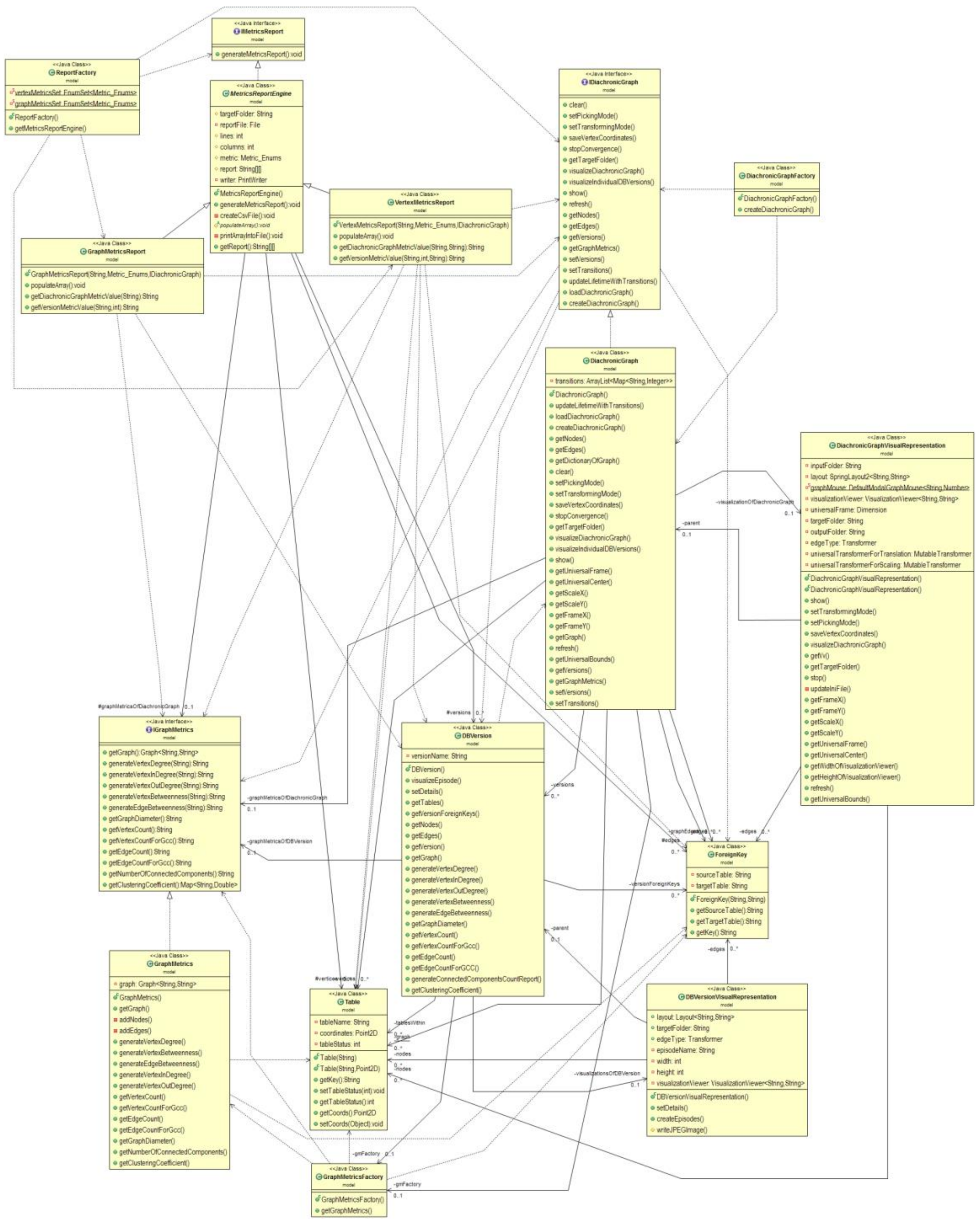
Figure 3.28 shows the class diagram of the *export* package.



<<Java Class>> ExportManagerFactory export
ExportManagerFactory()
createExportManager():IExportManager
<<Java Interface>> IExportManager export
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
<<Java Class>> ExportManager export
ExportManager()
createPowerPointPresentation(ArrayList<String>,String,String):void
createVideo(File):void
<<Java Class>> PowerPointGenerator export
targetWorkspace: String
presentation: String
width: int
height: int
PowerPointGenerator(String,String)
createPresentation(ArrayList<String>):void
initializePresentation(XMLSlideShow,ArrayList<String>):XMLSlideShow
appendSlideShow(String,XMLSlideShow):XMLSlideShow
setSlideTitle(String):String
<<Java Class>> VideoGenerator export
presentationSelected: File
filenameOfPresentation: String
FRAME_RATE: double
outputFilename: String
sourceFolder: String
VideoGenerator(File)
exportToVideo():void
extractPngFromPptx():void
createVideo():void
deleteGeneratedPng():void
setOutputFilename():void
setSourceFolder():void
convertToType(BufferedImage,int):BufferedImage

Figure 3.28 Updated Class Diagram of the Export Package

Figure 3.29 depicts the class diagram of the *model* package.



<<Java Interface>> IMetricsReport model
generateMetricsReport():void
<<Java Class>> ReportFactory model
vertexMetricsSet: EnumSet<Metric_Enums>
graphMetricsSet: EnumSet<Metric_Enums>
ReportFactory()
getMetricsReportEngine()
<<Java Class>> MetricsReportEngine model
targetFolder: String
reportFile: File
lines: int
columns: int
metric: Metric_Enums
report: String[][]
writer: PrintWriter
MetricsReportEngine()
generateMetricsReport():void
createCsvFile():void
populateArray():void
printArrayIntoFile():void
getReport():String[][]
<<Java Class>> VertexMetricsReport model
VertexMetricsReport(String,Metric_Enums,IDiachronicGraph)
populateArray():void
getDiachronicGraphMetricValue(String,String):String
getVersionMetricValue(String,int,String):String
<<Java Class>> GraphMetricsReport model
GraphMetricsReport(String,Metric_Enums,IDiachronicGraph)
populateArray():void
getDiachronicGraphMetricValue(String):String
getVersionMetricValue(String,int):String
<<Java Interface>> IDiachronicGraph model
clear()
setPickingMode()
setTransformingMode()
saveVertexCoordinates()
stopConvergence()
getTargetFolder()
visualizeDiachronicGraph()
visualizeIndividualDBVersions()
show()
refresh()
getNodes()
getEdges()
getVersions()
getGraphMetrics()
setVersions()
setTransitions()
updateLifetimeWithTransitions()
loadDiachronicGraph()
createDiachronicGraph()
<<Java Class>> DiachronicGraphFactory model
DiachronicGraphFactory()
createDiachronicGraph()
<<Java Class>> DiachronicGraph model
transitions: ArrayList<Map<String,Integer>>
DiachronicGraph()
updateLifetimeWithTransitions()
loadDiachronicGraph()
createDiachronicGraph()
getNodes()
getEdges()
getDictionaryOfGraph()
clear()
setPickingMode()
setTransformingMode()
saveVertexCoordinates()
stopConvergence()
getTargetFolder()
visualizeDiachronicGraph()
visualizeIndividualDBVersions()
show()
getUniversalFrame()
getUniversalCenter()
getScaleX()
getScaleY()
getFrameX()
getFrameY()
getGraph()
refresh()
getUniversalBounds()
getVersions()
getGraphMetrics()
setVersions()
setTransitions()
<<Java Class>> DiachronicGraphVisualRepresentation model
inputFolder: String
layout: SpringLayout2<String,String>
graphMouse: DefaultModalGraphMouse<String,Number>
visualizationViewer: VisualizationViewer<String,String>
universalFrame: Dimension
targetFolder: String
outputFolder: String
edgeType: Transformer
universalTransformerForTranslation: MutableTransformer
universalTransformerForScaling: MutableTransformer
DiachronicGraphVisualRepresentation()
DiachronicGraphVisualRepresentation()
show()
setTransformingMode()
setPickingMode()
saveVertexCoordinates()
visualizeDiachronicGraph()
getVv()
getTargetFolder()
stop()
updateIniFile()
getFrameX()
getFrameY()
getScaleX()
getScaleY()
getUniversalFrame()
getUniversalCenter()
getWidthOfVisualizationViewer()
getHeightOfVisualizationViewer()
refresh()
getUniversalBounds()
-visualizationOfDiachronicGraph 0..1
-parent 0..1
#graphMetricsOfDiachronicGraph 0..1
<<Java Interface>> IGraphMetrics model
getGraph():Graph<String,String>
generateVertexDegree(String):String
generateVertexInDegree(String):String
generateVertexOutDegree(String):String
generateVertexBetweenness(String):String
generateEdgeBetweenness(String):String
getGraphDiameter():String
getVertexCount():String
getVertexCountForGcc():String
getEdgeCount():String
getEdgeCountForGcc():String
getNumberOfConnectedComponents():String
getClusteringCoefficient():Map<String,Double>
-graphMetricsOfDiachronicGraph 0..1
-graphMetricsOfDBVersion 0..1
#versions 0..*
<<Java Class>> DBVersion model
versionName: String
DBVersion()
visualizeEpisode()
setDetails()
getTables()
getVersionForeignKeys()
getNodes()
getEdges()
getVersion()
getGraph()
generateVertexDegree()
generateVertexInDegree()
generateVertexOutDegree()
generateVertexBetweenness()
generateEdgeBetweenness()
getGraphDiameter()
getVertexCount()
getVertexCountForGcc()
getEdgeCount()
getEdgeCountForGCC()
generateConnectedComponentsCountReport()
getClusteringCoefficient()
-versions 0..*
-graphEdges #edges 0..*
-edges 0..*
<<Java Class>> ForeignKey model
sourceTable: String
targetTable: String
ForeignKey(String,String)
getSourceTable():String
getTargetTable():String
getKey():String
-versionForeignKeys 0..*
-parent 0..1
-edges 0..*
<<Java Class>> GraphMetrics model
graph: Graph<String,String>
GraphMetrics()
getGraph()
addNodes()
addEdges()
generateVertexDegree()
generateVertexBetweenness()
generateEdgeBetweenness()
generateVertexInDegree()
generateVertexOutDegree()
getVertexCount()
getVertexCountForGcc()
getEdgeCount()
getEdgeCountForGcc()
getGraphDiameter()
getNumberOfConnectedComponents()
getClusteringCoefficient()
#verticesWithin 0..*
<<Java Class>> Table model
tableName: String
coordinates: Point2D
tableStatus: int
Table(String)
Table(String,Point2D)
getKey():String
setTableStatus(int):void
getTableStatus():int
getCoords():Point2D
setCoords(Object):void
-tablesWithin 0..*
-graph 0..*
-nodes 0..*
-nodes
<<Java Class>> DBVersionVisualRepresentation model
layout: Layout<String,String>
targetFolder: String
edgeType: Transformer
episodeName: String
width: int
height: int
visualizationViewer: VisualizationViewer<String,String>
DBVersionVisualRepresentation()
setDetails()
createEpisodes()
writeJPEGImage()
-visualizationsOfDBVersion 0..1
-gmFactory 0..1
-gmFactory 0..1
<<Java Class>> GraphMetricsFactory model
GraphMetricsFactory()
getGraphMetrics()

Figure 3.29 Updated Class Diagram of the Model Package

3.6 Evaluation

In this section, we attempt to evaluate the software quality of ParmenidianTruth, after the refactoring process, using various metrics. This evaluation provides an overview of the enhancements that refactoring actions achieved in design level by comparing the values of the metrics before and after the refactoring procedure.

3.6.1 Abstractness – Instability Graph

In this step, we were interested in identifying how our modifications affected the packages of the tool. To this end, we used the *instability* and the *abstractness* metrics [Mart00] with the former metric used to reveal the effort required to make changes in one package and the latter representing the degree of the abstractness within each package. The instability metric assesses the degree of the violation of *Stable Dependencies Principle* (SDP), which defines as unstable a package that has many dependencies upon other packages. The violation of this principle results in creating a system that is not flexible to changes, since minor changes in one package can affect many others that depend upon it. The abstractness metric evaluates the abidance to *Stable Abstractions Principle* (SAP), which determines as stable a package that consists of many abstract classes and interfaces.

Terminology. The *instability* metric is given by the following equation

$$I = \frac{Ce}{Ca + Ce}$$

where I is the instability of the package, Ce the number of outgoing edges to packages upon which the package depends and Ca the number of incoming edges from packages that depend upon it. If $I = 0$, the corresponding package is independent and thus is considered as a stable package, whereas a package with $I = 1$ means that there are no any other packages that depend upon it and so the package is considered to be unstable since it only depends on other packages.

The *abstractness* metric is expressed as follows

$$A = \frac{Na}{Nc}$$

where A is the abstractness of the package, Na the number of the abstract classes and the interfaces included in the package and Nc the number of its classes. If $A = 0$, the package consists exclusively of concrete classes and the other packages that depend upon this package are prone to changes applied to each class of the package. On the other hand, if $A = 1$ the package comprises just a contract, a case which should be avoided due to the fact that a package is supposed to contain a set of modules that depend upon each other aiming at providing a single functionality.

At this point, we should mention that for the evaluation of the refactoring process we exploited Structure Analysis for Java (STAN) [Buga07], which is a tool that offers a set of code quality metrics.

Figures 3.30 and 3.31 depict these metrics for ParmenidianTruth before and after our refactoring actions. The horizontal axis represents the abstractness of the packages and the vertical axis their instability.

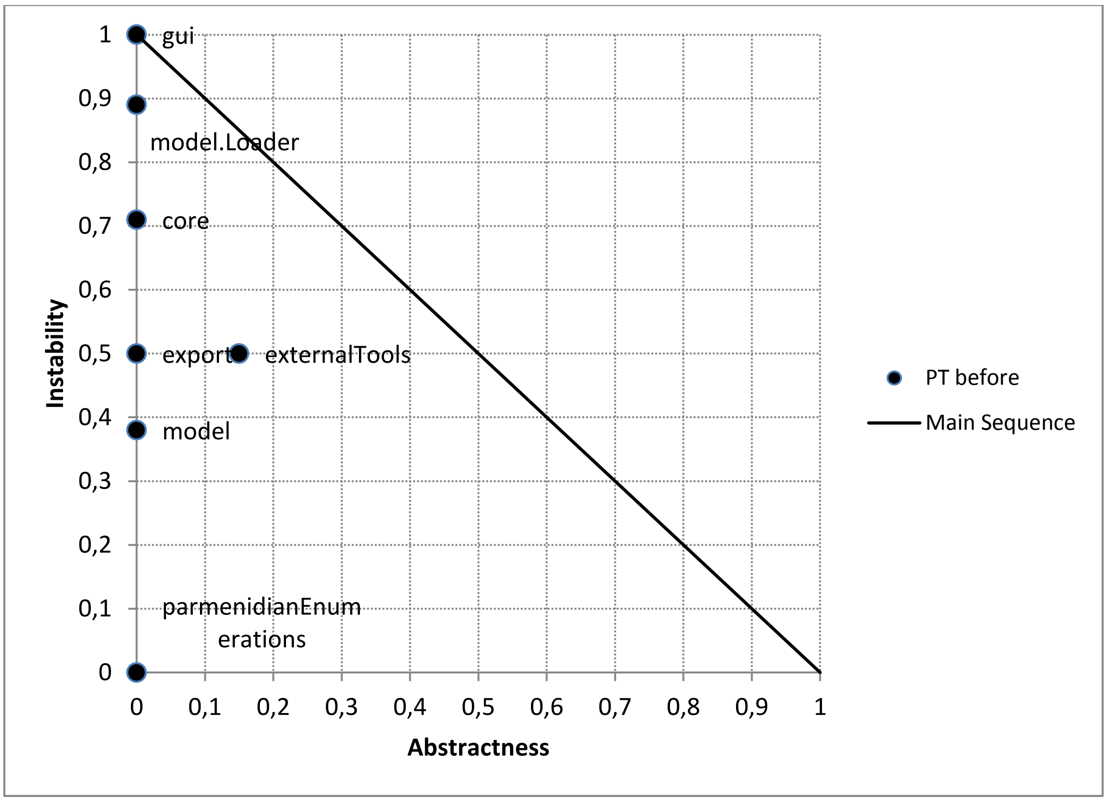


Figure 3.30 Abstractness-Instability Graph Before Refactoring Process

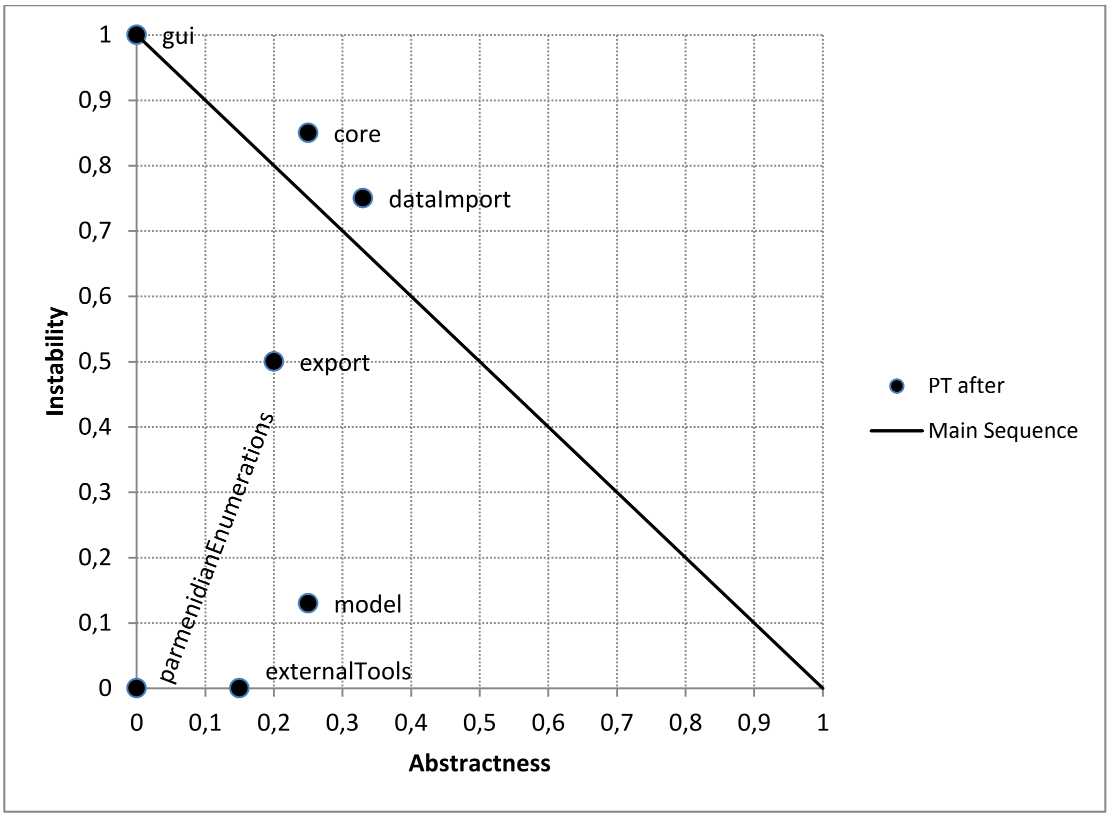


Figure 3.31 Abstractness-Instability Graph After Refactoring Process

It is worth mentioning that Figure 3.30 reveals the total absence of abstractions for the initial design of ParmenidianTruth tool, eliminating any possibility for extension, since it is difficult to predict the effects each modification in one package would create in the other ones. On the other hand, it is obvious in Figure 3.31 that the refactoring process increased the potentials for introducing new functionalities in ParmenidianTruth software without having to alter its subsystems to a large extent. This is feasible due to the addition of abstractions in almost all packages increasing their abstractness, combined with the reduction of the dependencies from concrete classes which decreases their instability.

3.6.2 Class Level Metrics

In a second approach concerning the improvements that our refactoring actions achieved in ParmenidianTruth tool, we assessed the quality of the classes of each package by using four metrics. Furthermore, we utilized the *number of methods*, the *number of fields*, the *Coupling Between Objects* (CBO) and the *Lack of COhesion of Methods* (LCOM).

Terminology. In [ChKe92], the *CBO* for a class is defined as the number of couples with other classes. In other words, an object is coupled to another one when it uses methods or instance variables of the other. The more coupled an object, the more sensitive to changes made in the parts that depends upon.

As for *LCOM*, we can define it as follows [ChKe92]:

Let C_1 is a class with n methods M_1, M_2, …, M_n and $\{A_i\}$ the set of class's attributes used by method M_i with $1 \leq i \leq n$. Let $P = \{(M_i, M_j) \mid A_i \cap A_j = \emptyset\}$ the number of pairs of methods that do not share attributes and $Q = \{(M_i, M_j) \mid A_i \cap A_j \neq \emptyset\}$ the number of pairs of methods that share at least one attribute. LCOM is defined as

$$LCOM = \begin{cases} P - Q, if\ P - Q \geq 0 \\ 0, otherwise \end{cases}$$

This metric gives us a notion of the degree of similarity of methods within a class, which means that the lower the value of this metric, the more cohesive the corresponding class.

The next charts represent the distribution of classes of each package with respect to the previous mentioned metrics. We present these metrics only for packages that underwent a set of changes during the refactoring process. This is the reason we excluded from this evaluation the *gui*, the *externalTools*, the *fileFilter* and the *parmenidianEnumerations* packages.

Figures 3.32, 3.33, 3.34 and 3.35 show the distribution of the classes of the *core* package with respect to the four metrics before and after the refactoring process. The black and gray colors denote ParmenidianTruth before and after the refactoring, respectively. In Figure 3.32, the horizontal axis corresponds to the number of methods in the *core* package and the vertical axis represents the percentage of the classes.

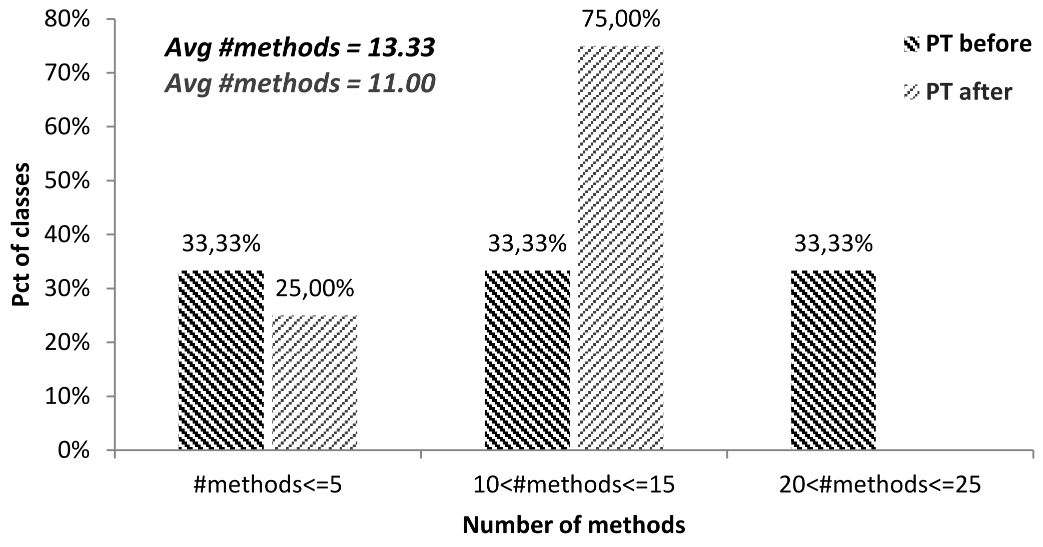


Figure 3.32 Distribution of Classes wrt Number of Methods (range) in the Core Package

From Figure 3.32 it is obvious that there is an elimination of the classes with more than 20 methods and a considerable increase in the percentage of classes that encompass from 10 to 15 methods.

In Figure 3.33, the distribution of classes with respect to the number of fields is depicted. The x – axis presents the number of fields in the *core* package and the y – axis the percentage of classes that include the corresponding number of fields.

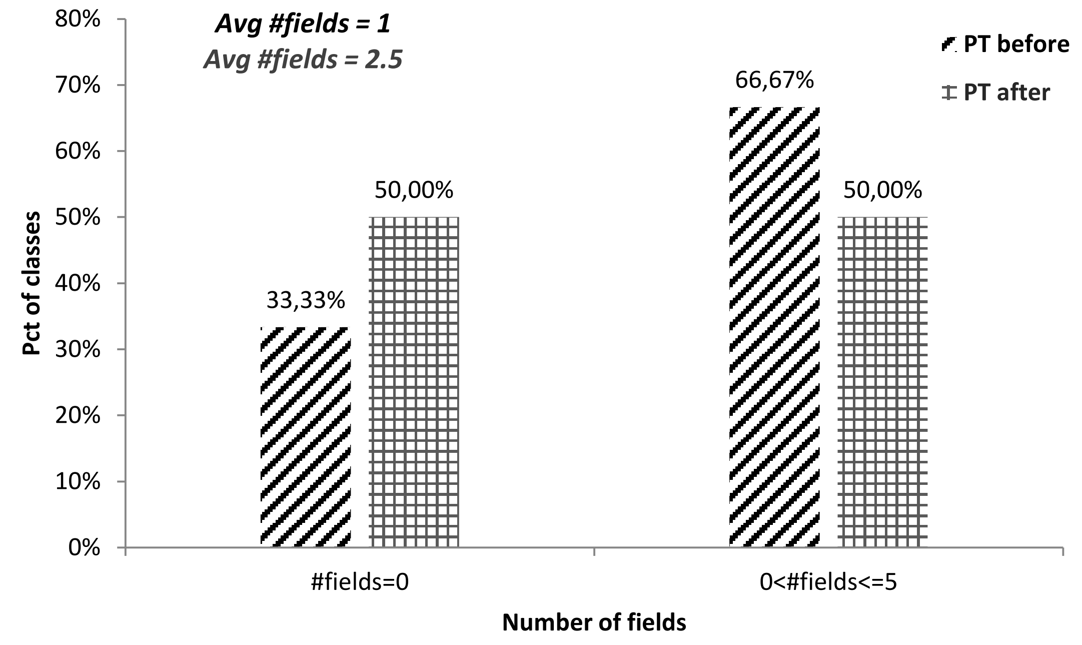


Figure 3.33 Distribution of Classes wrt Number of Fields (range) in the Core Package

As far as the number of fields is concerned, the refactoring process led to a balanced distribution between the modules that do not have any fields and these are the interfaces included in the *core* package and the remaining concrete classes.

Figure 3.34 contains the distribution of classes with respect to the Coupling Between Objects metric. The horizontal axis represents the values of the CBO metric and the vertical one the percentage of the classes in the *core* package.

The results for the Coupling Between Objects metric reveal a substantial reduction in the number of classes having dependencies in the range from 1 to 5 classes and the equally distribution of the percentage reduction in classes that do not have any couplings and those that depend from 6 to 10 classes.

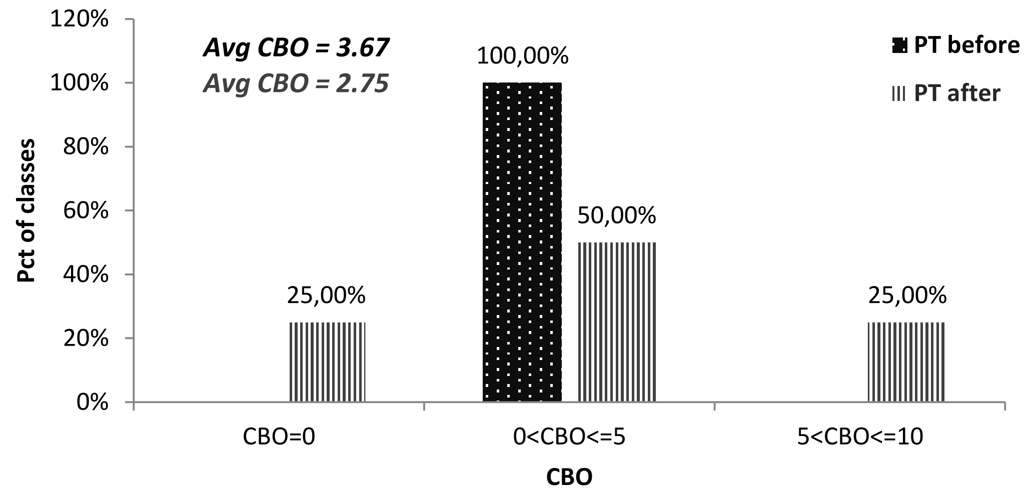


Figure 3.34 Distribution of Classes wrt CBO (range) in the Core Package

Figure 3.35 shows the spread of the classes over the values of the Lack of Cohesion metric. The x and y – axes correspond to the values of LCOM metric and the percentage of classes, respectively.

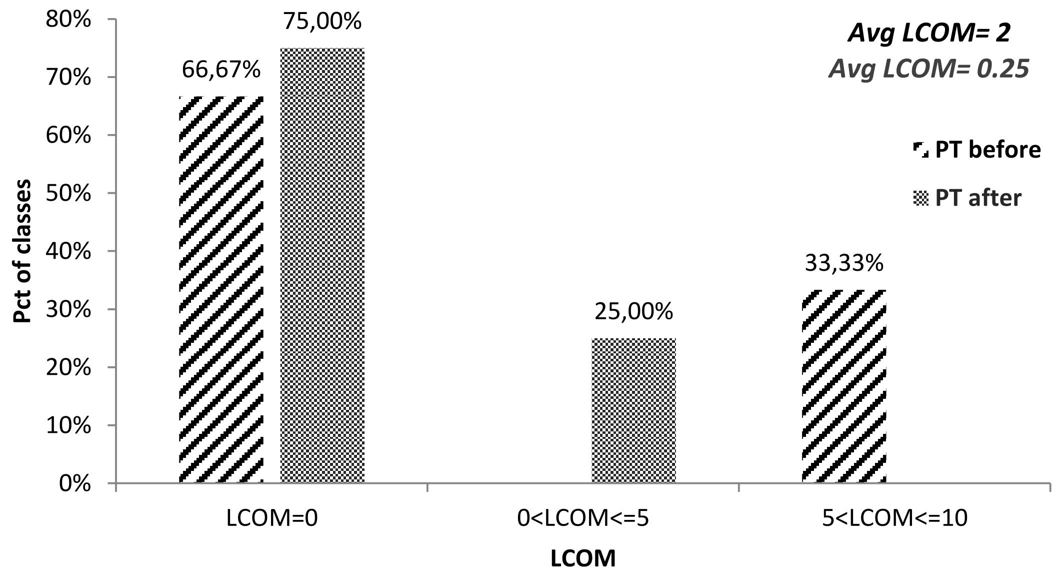


Figure 3.35 Distribution of Classes wrt LCOM (range) in the Core Package

We should mention that the Lack of Cohesion in the *core* package was completely removed in the range from 6 to 10 and increased nearly by 10% for the classes that are tightly cohesive.

Figures 3.36, 3.37, 3.38 and 3.39 present the distribution of the classes of the *export* package for the four metrics, before and after the refactoring procedure.

Figure 3.36 depicts the distribution of the classes with respect to the numbers of methods in the *export* package. The horizontal axis represents the number of methods and the vertical one the percentage of the classes.

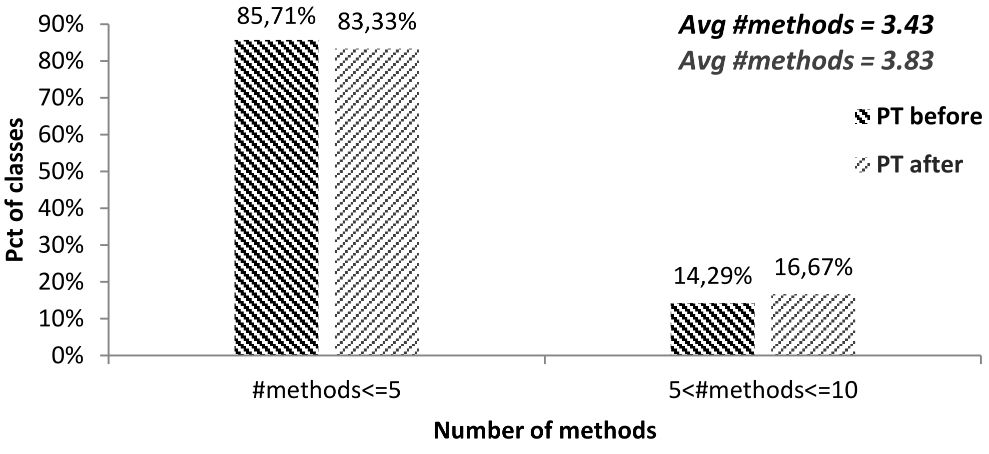


Figure 3.36 Distribution of Classes wrt Number of Methods (range) in the Export Package

Figure 3.37 shows how classes are distributed with reference to the number of fields in the *export* package. The x and y – axes correspond to the number of fields and the percentage of classes, respectively.

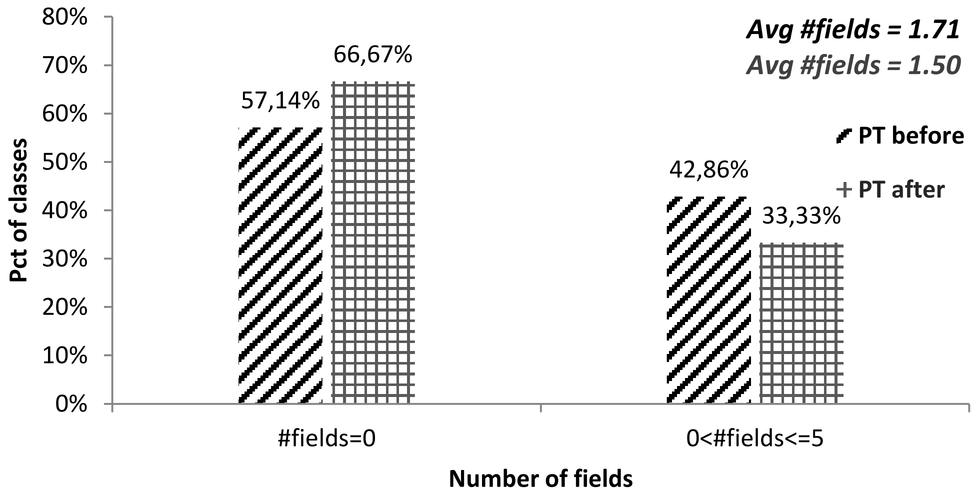


Figure 3.37 Distribution of Classes wrt Number of Fields (range) in the Export Package

Figure 3.38 shows the distribution of the classes with regard to the CBO metric in the *export* package. The horizontal axis includes the values of the CBO metric and the vertical one the number of classes expressed with reference to the total number of classes.

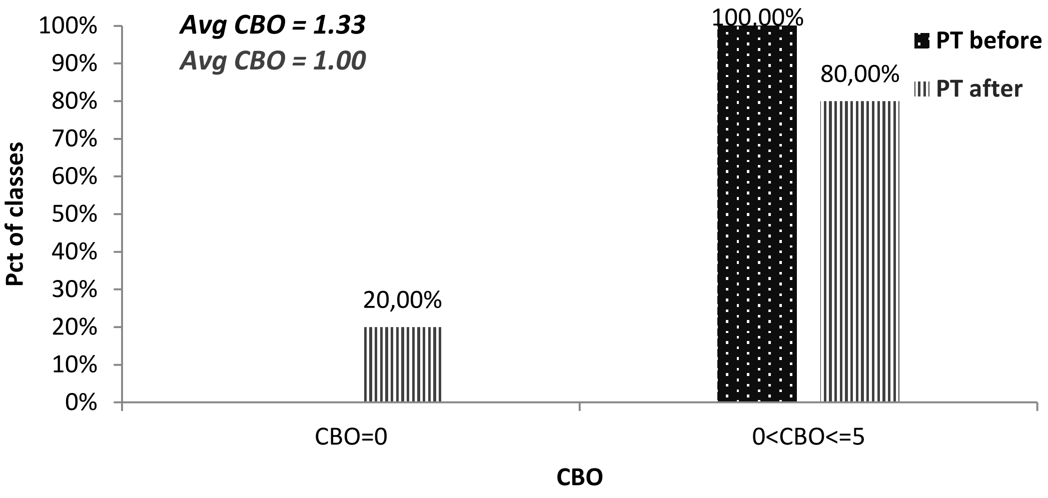


Figure 3.38 Distribution of Classes wrt CBO (range) in the Export Package

Figure 3.39 depicts the distribution of the classes with respect to the LCOM metric in the *export* package. The x and y – axes represent the values of the LCOM metric and the percentage of the classes, respectively.

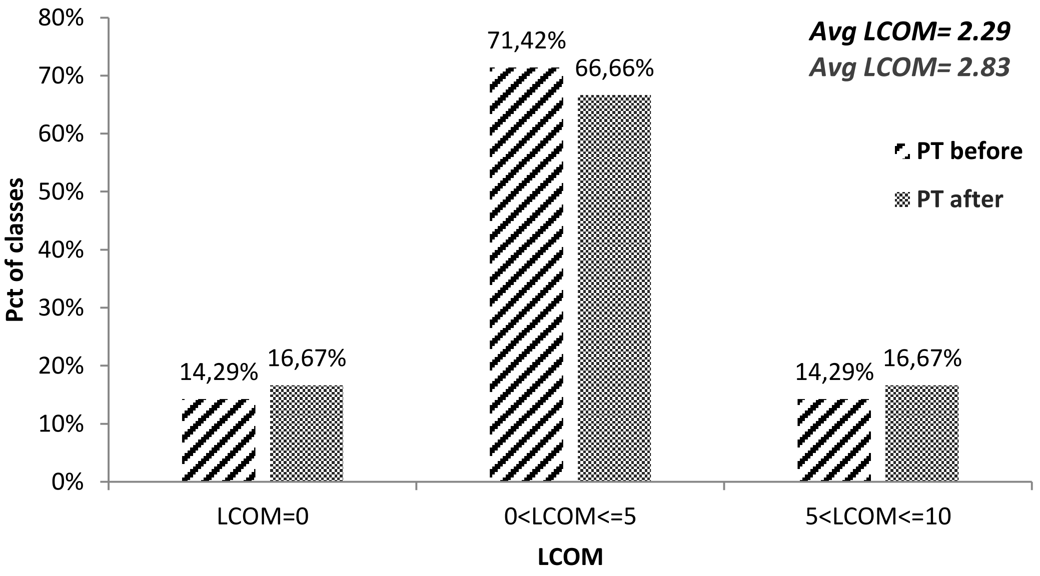


Figure 3.39 Distribution of Classes wrt LCOM (range) in the Export Package

To summarize the results for the *export* package, there were some minor fluctuations in the number of methods, the number of fields and the values of LCOM and a 20% reduction in the CBO metric in the range from 1 to 5.

In Figures 3.40, 3.41, 3.42 and 3.43 the number of classes with regard to the metrics for the *model* package is shown.

Figure 3.40 shows the distribution of the classes with respect to the number of methods in the *model* package. The horizontal axis corresponds to the number of methods and the vertical axis represents the relative number of the classes.

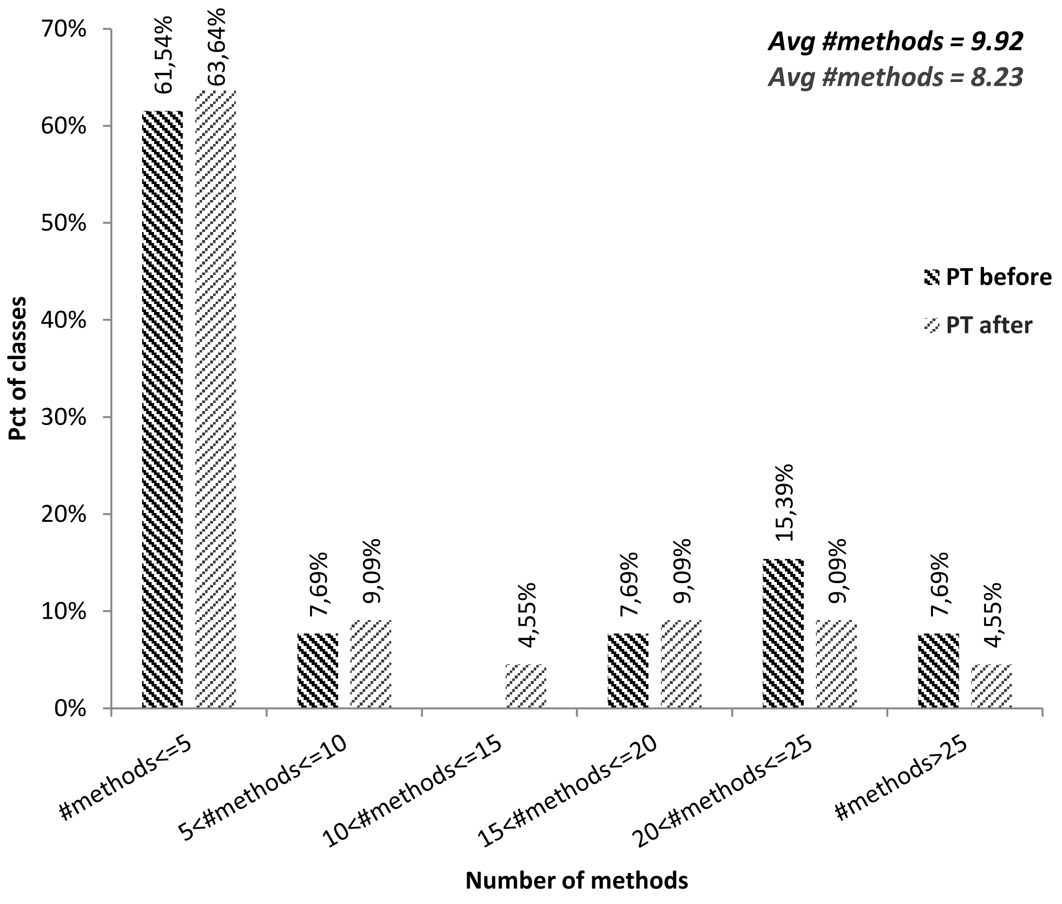


Figure 3.40 Distribution of Classes wrt Number of Methods (range) in the Model Package

In Figure 3.41, the distribution of the classes with reference to the number of fields in the *model* package is presented. The x and y – axes denote the number of fields and the percentage of the classes, respectively.

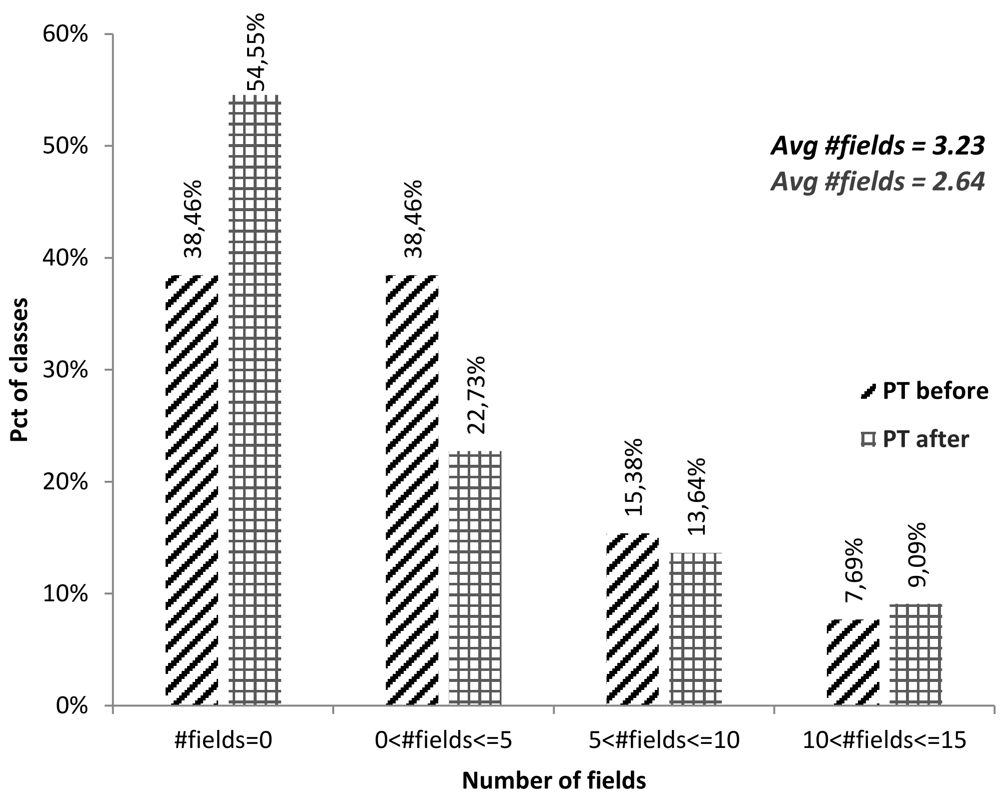


Figure 3.41 Distribution of Classes wrt Number of Fields (range) in the Model Package

Figure 3.42 shows how the classes are divided with respect to the CBO metric in the *model* package. The horizontal axis includes the values of the CBO metric and the vertical one the relative number of the classes.

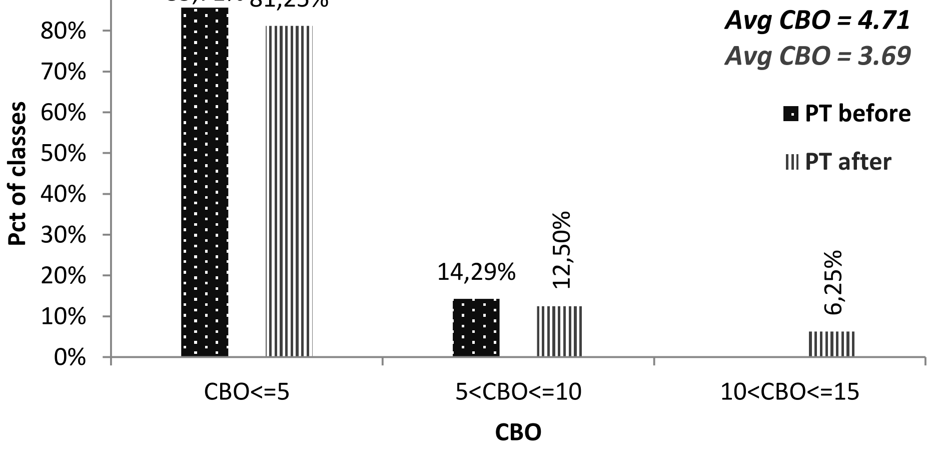


Figure 3.42 Distribution of Classes wrt CBO (range) in the Model Package

In Figure 3.43, the distribution of the classes with regard to the LCOM metric in the *model* package is presented. The x and y – axes represent the values of the LCOM metric and the percentage of the classes, respectively.

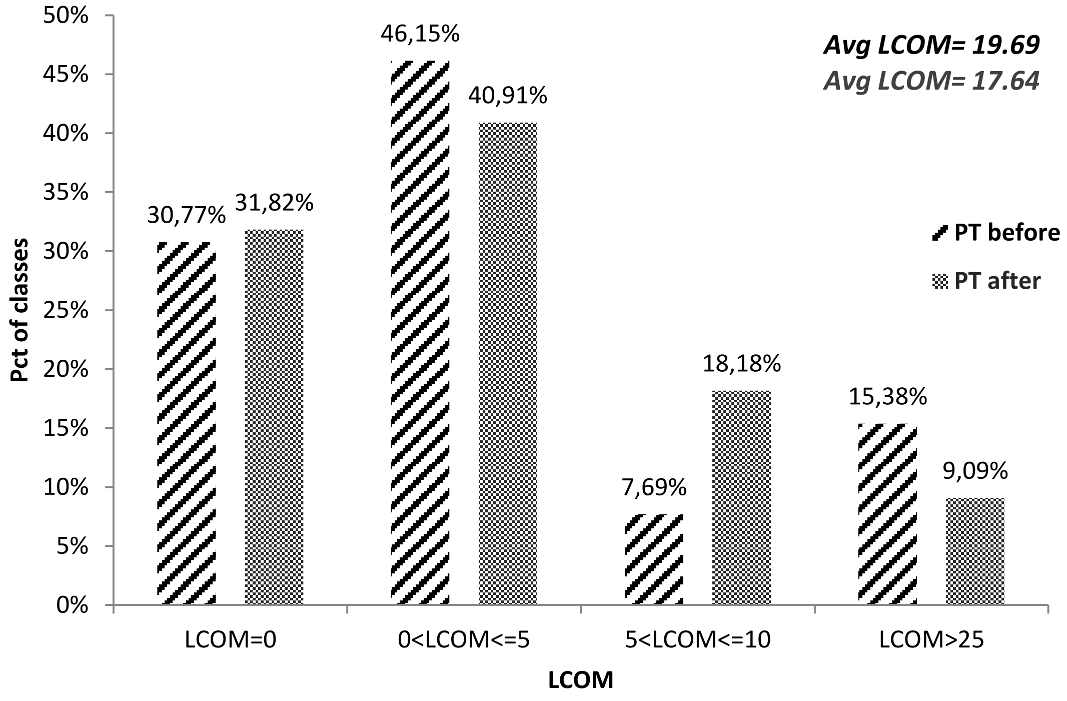


Figure 3.43 Distribution of Classes wrt LCOM (range) in the Model Package

From the previous figures, we can claim that we managed to reduce the number of classes that included more than 20 methods and increase the number of modules without fields by introducing interfaces. However, there appears a small increase in the number of classes with CBO greater than 10 and a significant growth in the number of classes with LCOM in the range of 5 to 10.

The Figures 3.44, 3.45, 3.46 and 3.47 depict the distribution of the classes of the *dataImport* package concerning the four metrics.

In Figure 3.44, the division of the classes with reference to the number of the methods in the *dataImport* package is shown. The x – axis correspond to the number of the methods and the y – axis to the percentage of the classes.

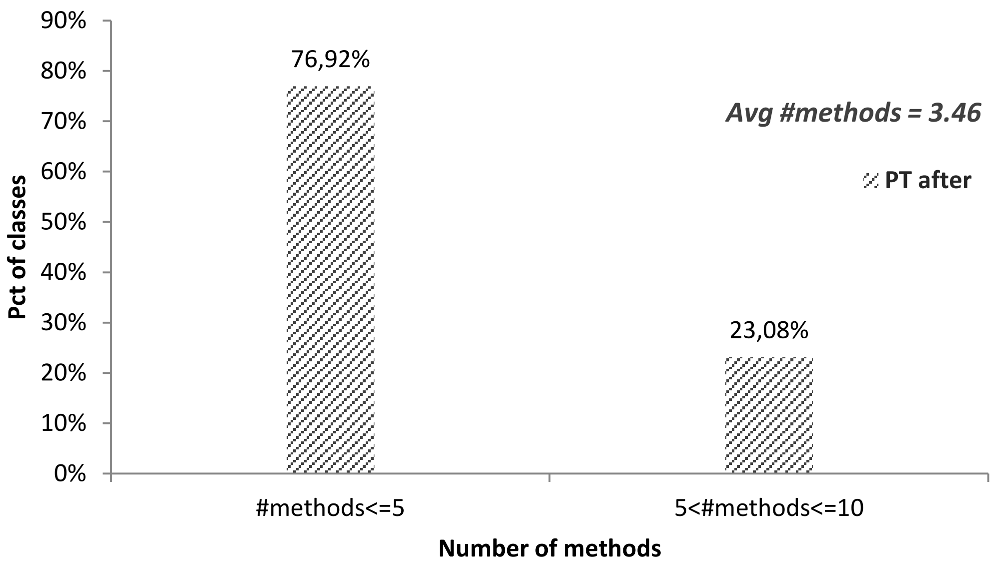


Figure 3.44 Distribution of Classes wrt Number of Methods (range) in the DataImport Package

Figure 3.45 depicts the distribution of the classes with respect to the number of the fields in the *dataImport* package. The horizontal axis represents the number of the fields and the vertical one the relative number of the classes.

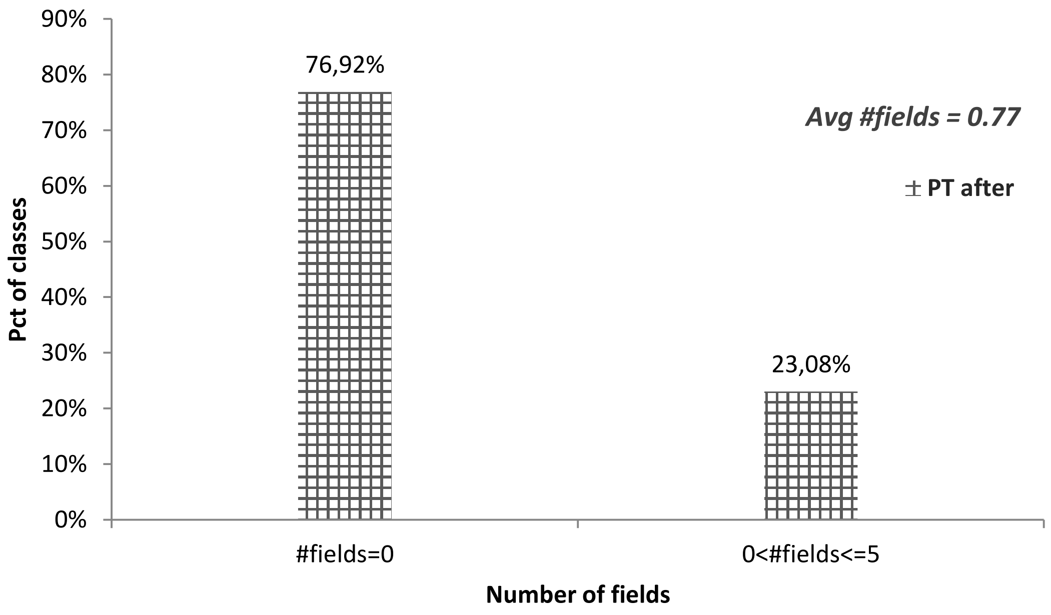


Figure 3.45 Distribution of Classes wrt Number of Fields (range) in the DataImport Package

In Figure 3.46, the distribution of the classes with respect to the CBO metric in the *dataImport* package is presented. The x and y – axes correspond to the values of the CBO metric and the percentage of the classes, respectively.

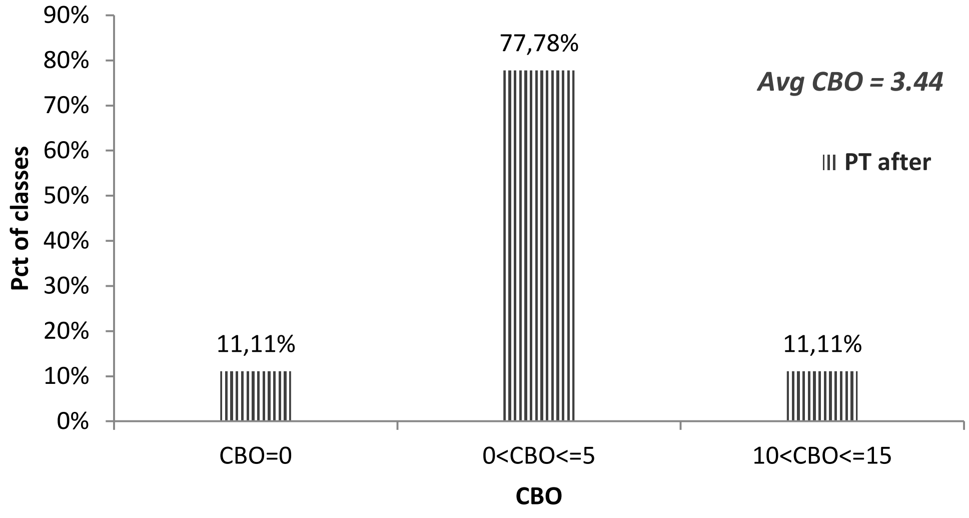


Figure 3.46 Distribution of Classes wrt CBO (range) in the DataImport Package

Figure 3.47 depicts the distribution of the classes with regard to the LCOM metric in the *dataImport* package. The horizontal axis represents the values of the LCOM metric and the vertical one the relative number of the classes.

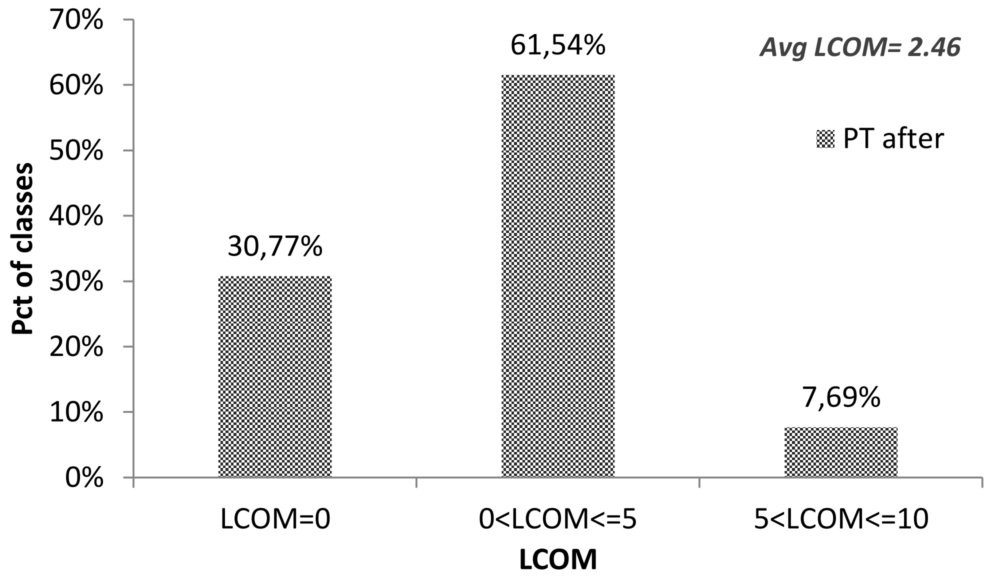


Figure 3.47 Distribution of Classes wrt LCOM (range) in the DataImport Package

To synopsize the improvements achieved in the modules of the Parmenidian Truth tool through the refactoring process, we give a brief description of the enhancements in the following figure.

Subsystem	**# methods**	**# fields**	**CBO**	**LCOM**
core	• 30% less classes with more than 20 methods • 40% more classes with 10 to 15 methods	• Balanced distribution between classes with no fields and those with less than 5 fields	• 50% less classes with CBO in the range 1 to 5	• 30% less classes with LCOM in the range 6 to 10 • 10% more tightly cohesive classes
export	• Minor changes wrt the number of methods	• 9% less classes with at least 1 field	• 20% less classes with CBO in the range 1 to 5	• Small changes wrt the LCOM metric
model	• Nearly 10% less classes with more than 20 methods	• 16% more modules with no fields (interfaces)	• A small increase (6%) in the number of classes with CBO in the range 11 to 15	• 10% more classes with LCOM in the range 6 to 10
dataImport	• 77% of the modules with less than 5 methods	• 77% of the modules with no fields	• 78% of the modules with CBO in the range 1 to 5	• 62% of the modules with LCOM in the range 1 to 5

Figure 3.48 Summary of the Improvements of the Refactoring Process

3.7 Summary of Refactoring Results

Concluding this chapter, we summarize our actions and results as follows:

- We have eliminated violations of the Single – Responsibility Principle by extracting methods from classes that include more than one responsibility.

- We have increased Parmenidian Truth's expandability and immunity to modifications by introducing a set of APIs.
- We have eliminated duplicated code by utilizing the *template method* design pattern.
- We have increased cohesion of methods by identifying misplaced methods and assigning them in new classes.
- We have reduced the complexity of the code and facilitating its evolvement by removing redundant components.
- We have complied with code conventions by identifying related violations and making the required adjustments.
- We have verified the correctness of our modifications by creating a test case for each of the modules we either modified or added.

CHAPTER 4.

TABLE TOPOLOGY AND EVOLUTION

As already discussed in previous sections, we are equipped with both the model and the tool support to treat database schemata as graphs, in which nodes and edges represent the tables of the dataset and the foreign key constraints between tables, respectively. Given that, we can exploit the information on the position of a table in the graph to see whether such information can be correlated to the evolution activity of the table. We use the term "table topology" in its etymological sense, much like as it is also used when referring to network topology, meaning the pattern of edges surrounding nodes.

In this chapter, our main objective is to study the topology of the tables in 5 open-source datasets and identify possible patterns concerning the evolution of the tables with reference to the topological categories they belong to. In the first section, we introduce the datasets used in our study and describe the preprocessing actions taken to eliminate data that would not help us arrive at valid conclusions. The second section presents the distribution of the tables over their in-, out- and total degrees, information exploited in the following section to define the topological categories. The third section includes, apart from the determination of the topological categories, a set of classification rules applied to classify tables in these categories. The fourth section examines the evolution of tables with respect to their topological categories and other properties including tables' duration, survival potential, version of birth, update activity and size change. Finally, in the last section we summarize the main conclusions derived from our study and evaluate the extent to which our initial research questions are addressed.

4.1 Experimental Setup

In this section we present the experimental setup of our study. First, we start with the main features of the 6 open-source datasets utilized in our study. Next, we report on the preprocessing actions that we have taken in order to exclude information that is considered to be useless in the context of this research. At this point, it is worth mentioning that all the graph–related metrics we use to study the schema and its evolution are obtained via the Parmenidian Truth tool whose main functionalities were described in more detail in Chapter 3.

4.1.1 Datasets

The datasets concerning this study support projects from different domains and have a common feature, which is the availability of their source files that allows us to conduct a research into the evolution of their structure. Figure 4.1 synopsizes for each dataset the information about the number of the tables and the foreign keys at the first version, the last version and the Diachronic Graph. The statistics concerning the Diachronic Graph express the total number of unique tables or foreign keys that exist over the period that database schema's evolution is examined.

Atlas Trigger is the dataset that supports the *ATLAS* experiment which is one of the four experiments conducted at the Large Hadron Collider in the facilities of CERN in Geneva, Switzerland. The schema history of Atlas Trigger consists of 85 versions including 88 tables and 88 foreign key constraints. It started its life with 56 tables and 61 foreign keys and ended up with 73 tables and 63 foreign keys. The growth of tables as well as that of foreign keys between the first and the last version of its life is positive, reaching the values of 30% and 3%, respectively.

BioSQL is a generic relational model for storing sequences, features and ontologies derived from different sources aiming at facilitating the interoperability of projects implemented by the Open Bioinformatics Foundation (OBF). Our study concerns 47 versions that include 45 tables and 79 foreign keys. The first version includes 21 tables and 17 foreign keys and the last one 28 tables and 43 foreign keys resulting in a growth of 33% and 153% for tables and foreign keys, respectively.

The *Cern Advanced STORage* (*CASTOR*) manager is the next database whose schema evolution is being examined in the current study and its' main goal is to store and provide remote access to physics data. Its' 194 versions comprise 91 tables and 13 foreign key constraints, with the corresponding numbers in the first and last version to be 62, 6 and 74, 10 respectively. The growth in the number of tables and foreign keys is 19% and 67% in the order given.

The *Enabling Grids for E-sciencE* (*EGEE*) project provided a world-wide infrastructure for e-science, allowing the exploitation of its computer power and the data storage capacity by numerous research groups around the world. For the period examined, this dataset consists of 17 versions including 12 tables and 6 foreign keys, starts its life with 6 tables and 3 foreign keys and eventually finishes up with 10 tables and 4 foreign keys. The respective growth in the number of tables is 67% and in the number of foreign key constraints 33%.

SlashCode is a content management system that initially used to support Slashdot, a social news website. Its' 399 versions encompass 126 tables and 47 foreign keys, with the first version comprising 42 tables but no foreign keys as it is also the case for the last version where the number of tables reaches the value of 87. The corresponding growth rate of the tables between the first and the final version is 107%.

Zabbix is an open – source monitoring software for networks, operating systems and applications, which comprises 160 versions with 58 tables and 38 foreign key constraints. The originating version of Zabbix includes 15 tables and 10 foreign keys and the last one 48 tables and 2 foreign keys resulting in a growth rate of 220% for tables and -80% for foreign keys.

It is noteworthy that in all the datasets the growth rate of tables is positive, a trend that also holds for foreign keys, with the exceptions of Zabbix and

SlashCode, where in the former there appears a significant decline in the number of foreign keys and in the latter a total absence of foreign keys in the first and the last versions. Figure 4.2 depicts the growth rate of tables and foreign keys for each of the aforementioned datasets.

Datasets	Versions	Tables @start	Tables @end	Tables @DG	Tables Growth	FKs @start	FKs @end	FKs @DG	FKs Growth
Atlas	85	56	73	88	30,4%	61	63	88	3,3%
BioSQL	47	21	28	45	33,3%	17	43	79	152,9%
Castor	194	62	74	91	19,4%	6	10	13	66,7%
Egee	17	6	10	12	66,7%	3	4	6	33,3%
Slashcode	399	42	87	126	107,1%	0	0	47	0,0%
Zabbix	160	15	48	58	220,0%	10	2	38	-80,0%

Figure 4.1 Statistics for the datasets used in our study, [VKZZ17]

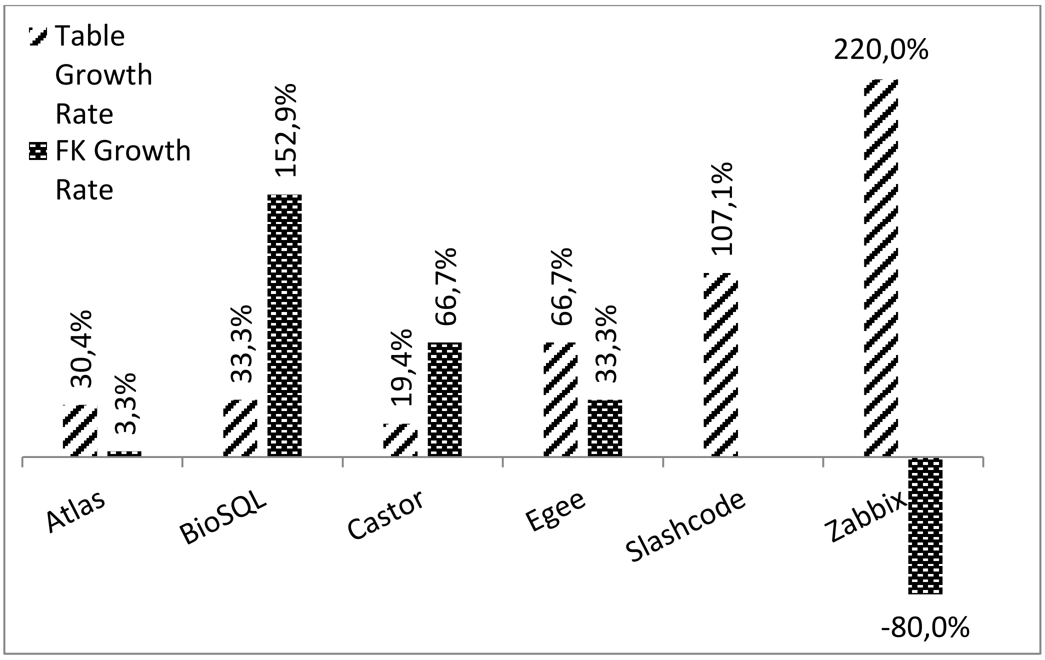
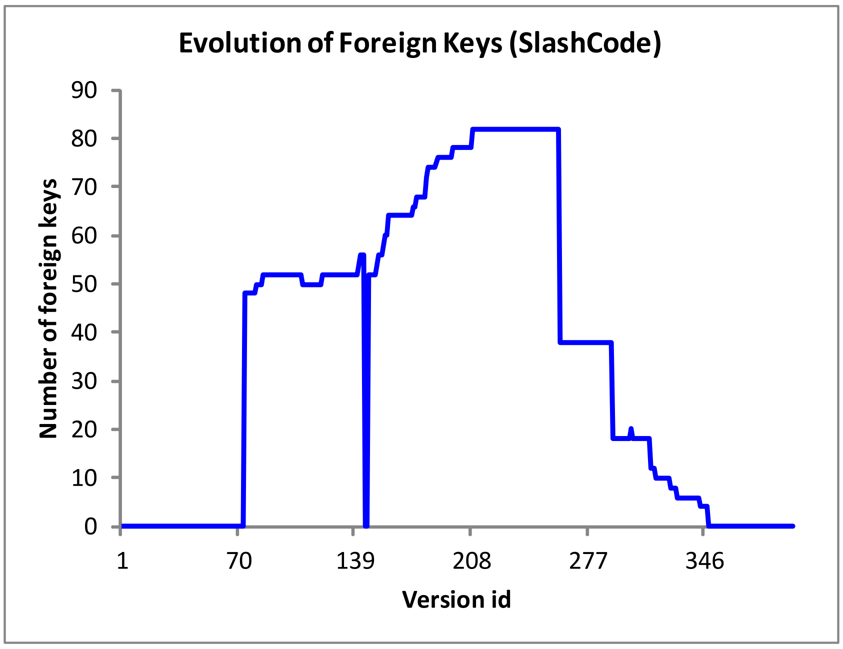


Figure 4.2 Growth Rate of Tables and Foreign Keys

4.1.2 Data Preprocessing

In this subsection we discuss the interventions we performed to the collected data, along with decisions taken to aid the extraction of valid conclusions. As already explained in [VKZZ17], two of the datasets, SlashCode and Zabbix, demonstrate the explicit removals of foreign keys from the schema, with the former also introducing foreign keys late in the schema history. We have decided to omit the periods where foreign keys were massively absent from the schema, since no table could possibly have any topological properties during these periods. Figure 4.3 depicts the evolution of foreign keys in these datasets.



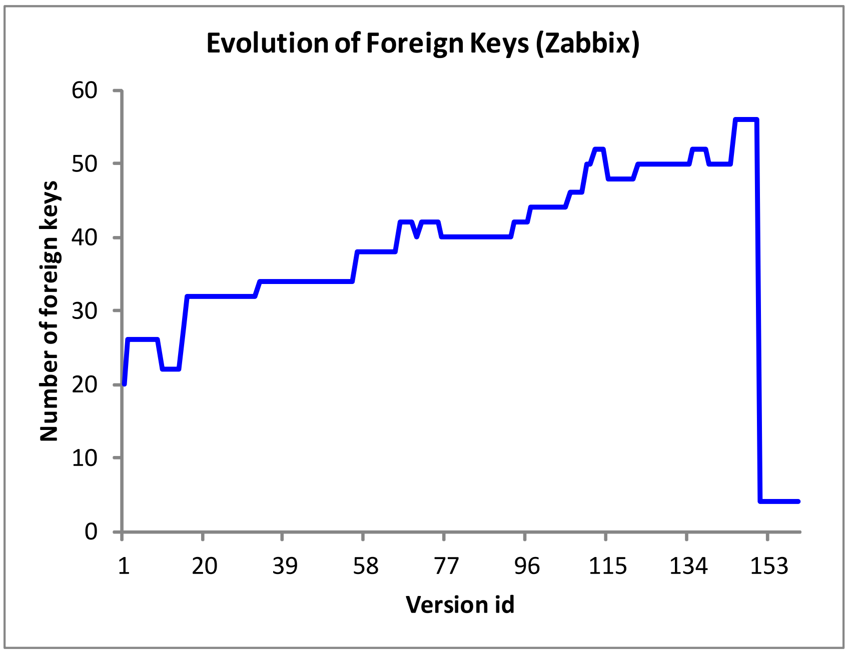


Figure 4.3 Evolution of Foreign Keys in SlashCode and Zabbix

In case of the SlashCode dataset depicted in the upper part of Figure 4.3, we distinguish the first 74 versions with no foreign keys as well as the interval after the version 260 after which we observe a continuing decrease in the number of foreign keys until the last version examined. As a result, we opted for limiting our study in the interval bounded by versions 74 and 260.

In a similar way, we examined the evolution of foreign keys in the Zabbix dataset and identified a steep decline in the number of foreign keys after version 150. Thus, we constrain our research in the period defined by versions 1 and 150.

4.2 Distribution of Tables over Degrees

As already mentioned in the introduction of this chapter, our main goal is to study the evolution of the tables with respect to the topological categories they belong to. Thus, prior to specifying the categories, it is vital to understand and obtain a comprehensive overview of the distribution of tables over the total degrees, in-degrees and out-degrees at the Diachronic Graph. Having done that, we will be able to assign the tables in the corresponding categories and study their evolution throughout their existence in the respective database schemata.

Figure 4.4 presents the distribution of the tables over their total degrees at the Diachronic Graph for the 6 studied datasets. The graphical part provides us with some interesting insights about the breakdown of tables over degrees summarized as follows:

- In 3 out of the 6 datasets, we encounter a substantial majority of zero-degree tables that in all cases surpasses the half of the respective total number of tables.
- In 4 out of the 6 datasets, there appears a decrease in the number of tables as the degree increases. This pattern, which is described in [VKZZ19] as a *monotone decrease* pattern, is the case for all the datasets with the exceptions of Atlas and BioSQL.
- Atlas and BioSQL present a different behavior, with the former following the so-called *battleship pattern* [VKZZ19], which starts with an increase in the number of including tables of degree from 0 to 2 followed by a significant decrease in the percentages of tables of higher degrees. On the contrary, the latter dataset demonstrates a "balanced" distribution of its tables among the different degrees with the majority of tables clustered in the degrees of 1 and 2.

	Degree @DG					
Datasets	0	1	2	3	>= 4	**Total**
Atlas	11	25	35	7	10	**88**
BioSQL	5	15	15	6	4	**45**
Castor	75	8	6	2	0	**91**
Egee	6	2	2	2	0	**12**
SlashCode	90	21	7	1	7	**126**
Zabbix	23	15	13	2	5	**58**

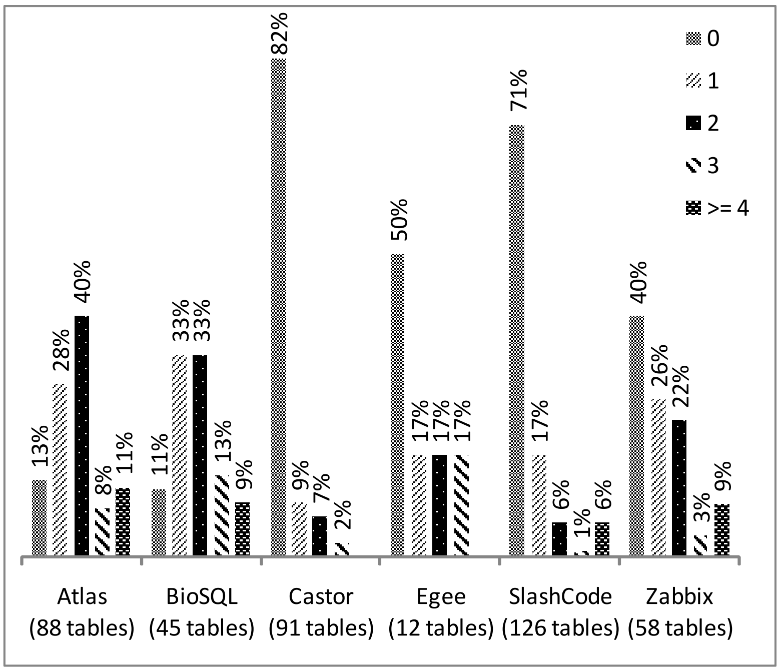


Figure 4.4 Distribution of Tables over Total Degrees

The distribution of the tables over their total degrees at the Diachronic Graph offered us the first useful information which is the strong presence of tables that have no references to other tables throughout their entire lives.

Our next step concerns a more in-depth analysis of tables' topology which will facilitate the process of defining the different table categories based on their in- and out-degrees at the Diachronic Graph. Figure 4.5 depicts the distribution of the tables of the datasets studied over their in-degrees.

The most intriguing observations concerning the breakdown of tables per in-degree at the Diachronic Graph are outlined in the following list:

- The tables with zero in-degree are the dominating ones, accounting for at least the 55% of the overall table population. Furthermore, in all the datasets, the number of tables in the "zero in-degree" bucket is an absolute majority, and frequently, a very large one.
- The trend for decreasing numbers of tables as the in-degree increases is also present in this breakdown and it holds in all the datasets. We should clarify that the increasing percentages of tables of in-degree higher than three are due to the aggregation nature of this category and this is the reason why the decrease in the number of tables is not shown as monotone in Figure 4.5.
- The tables with in-degree greater than 2 constitute a small minority that corresponds to values less than 4% in the datasets Castor, Egee and SlashCode. Compared to those datasets, Atlas, BioSQL and Zabbix encompass more tables of high in-degree, though the respective percentages do not exceed the value of 15%.

In a nutshell, we notice that few tables ever get an incoming edge and the probability of having more incoming edges monotonically decreases with the in-degree.

	In Degree @DG					
Datasets	0	1	2	3	>= 4	**Total**
Atlas	48	18	11	4	7	**88**
BioSQL	30	6	2	1	6	**45**
Castor	81	8	1	1	0	**91**
Egee	8	2	2	0	0	**12**
SlashCode	114	4	3	0	5	**126**
Zabbix	42	7	4	2	3	**58**

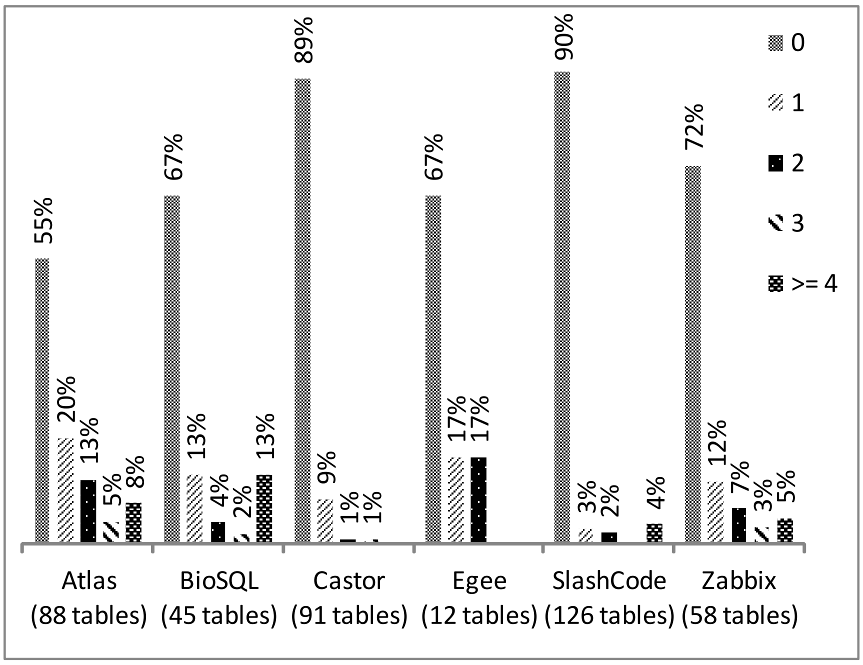


Figure 4.5 Distribution of Tables over In-Degrees

Figure 4.6 shows the distribution of the tables of the 6 datasets with reference to their out-degrees at the Diachronic Graph. As far as this distribution is concerned, we can make the following comments:

- Apart from Atlas and BioSQL, all the other datasets present a strong tendency towards the zero out-degree, a behavior similar to that encountered in the in-degree distribution but with a more moderate intensity here.
- The declining numbers of tables while out-degree increases are more obvious in the last three datasets, in contrast to the first two which concentrate a significant number of tables in the out-degrees of value 1

and 2. Especially, the tables of out-degree 2 account for the one third of the total table population in both datasets. After manual inspection, we attribute this phenomenon to the existence of several N:M relationships, modeled via tables of out-degree exactly equal to 2.

- As for the tables of out-degree higher than 2, they represent a small population in all the datasets excluding that of BioSQL. Compared to tables of in-degree higher than 2, tables of high out-degree are less and this can be attributed to the presence of lookup tables which attract a high number of incoming edges from other tables.

Out Degree @DG

Datasets	0	1	2	3	>= 4	**Total**
Atlas	43	14	28	0	3	**88**
BioSQL	7	12	14	9	3	**45**
Castor	83	3	5	0	0	**91**
Egee	7	4	1	0	0	**12**
SlashCode	95	20	8	2	1	**126**
Zabbix	32	15	10	1	0	**58**

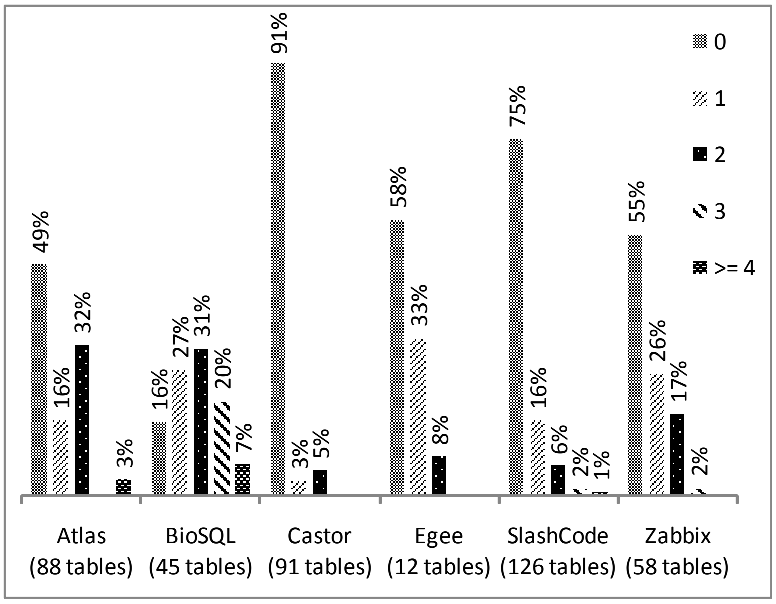


Figure 4.6 Distribution of Tables over Out-Degrees

Overall, we observe that in 4 of the 6 datasets the number of tables with out-degree in the range from 1 to 2 is higher compared to the respective number in the distribution over in-degrees. We should also stress the sparse population of tables with out-degree higher than 2, with the exception of

BioSQL, in which high out-degree tables are more than those with high in-degree.

4.3 Table Topological Categories

After having acquired a general overview of how tables are spread with respect to their in- , out- and total degrees at the Diachronic Graph, we now shift our focus to the combination of in- and out- degrees in order to define the distinctive categories utilized for studying tables' evolution with reference to the topological categories.

4.3.1 Definition of Topological Categories

In this subsection, we present the distinctive topological categories of tables based on their references to and from other tables. Figure 4.7 depicts the distribution of tables over the combination of their in- and out-degrees at the Diachronic Graph for the 6 datasets.

In-Degree @DG	Out-Degree @ DG	Atlas	BioSQL	Castor	Egee	SlashCode	Zabbix
0	0	11	2	75	6	90	23
≠0	0	32	5	8	1	5	9
0	≠0	37	28	6	2	24	19
≠0	≠0	8	10	2	3	7	7
Total		**88**	**45**	**91**	**12**	**126**	**58**

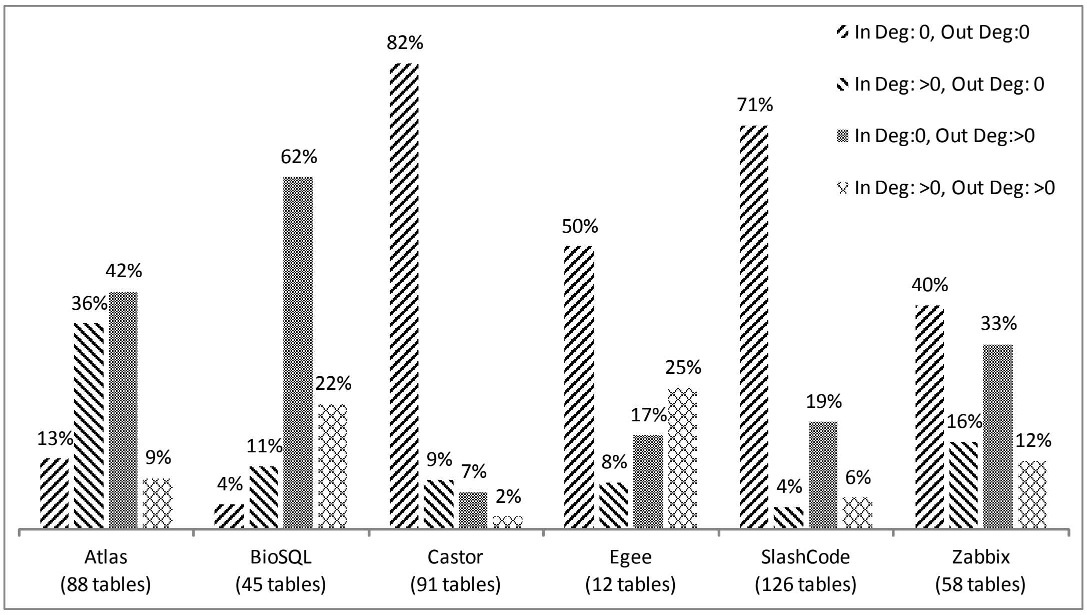


Figure 4.7 Breakdown of tables wrt In- and Out-Degrees at the Diachronic Graph

In the sequel, we introduce the different topological categories, which are determined on the basis of the topology of the Diachronic Graph.

The most obvious information portrayed in Figure 4.7 is the strong presence of tables with no inciting edges in 4 of the 6 datasets. Moreover, in two of these datasets, namely Castor and SlashCode, zero degree tables constitute an overwhelming majority. Given that our interest concerns the evolution of tables with respect to the graph topology, we concluded that tables without any references would not provide us with useful insights answering our research questions. Due to this, we will frequently accompany the statistical analyses with extra frequency tables where these tables, which from now on we will call *isolated*, are omitted and the respective percentages are counted over the set of tables with at least one inciting edge.

The next category consists of tables with no incoming references and at least one outgoing foreign key. This category of tables, which includes populations varying from 7% to 62%, is identified by the label *source* since the tables contained have only references to other tables.

The third category includes tables with only incoming references, so we distinct them with the label *lookup*. In the 6 datasets, there is a small group of tables that lie in this category, not exceeding the value of 36%, but we consider them to occupy a key role in a database's schema as they carry valuable information exploited by other tables, so it is worth studying their evolution as a standalone group.

The last two categories contain tables that have both in- and out-degrees. Although their population would not justify their division into two discrete groups, we assumed that there might be divergence between the tables of the two categories with respect to the nature of their role. The first of these two categories includes tables with in- and out-degrees equal to 1 and we use the term *chain link* for the participating tables, due to the fact that they operate as intermediate nodes in the topology of the Diachronic Graph. The second category, which encompasses tables with total degree greater than 2 and with both incoming and outgoing references, is defined with the term *mini-hub* since the tables included are neither fountains nor sinks of the graph, and thus, they are hub nodes in any possible path of the graph. Figure 4.8 presents the distribution of the tables within these two categories.

The overall population of tables included in the last two categories ranges between 2% and 25% of the population of their datasets, with each dataset not containing more than 10 such tables. Figure 4.8 demonstrates that the *mini-hub* category is the superior one in 5 of the 6 datasets, with the corresponding populations ranging from 57% to 100% with respect to the total number of tables included. Over the course of our study, and specifically in the phase of assigning a single label to each table, we realized that the number of tables included in the *chain link* category was too small and as a result they were absorbed by the *mini-hub* class, forming a unified category identified by the label *internal*.

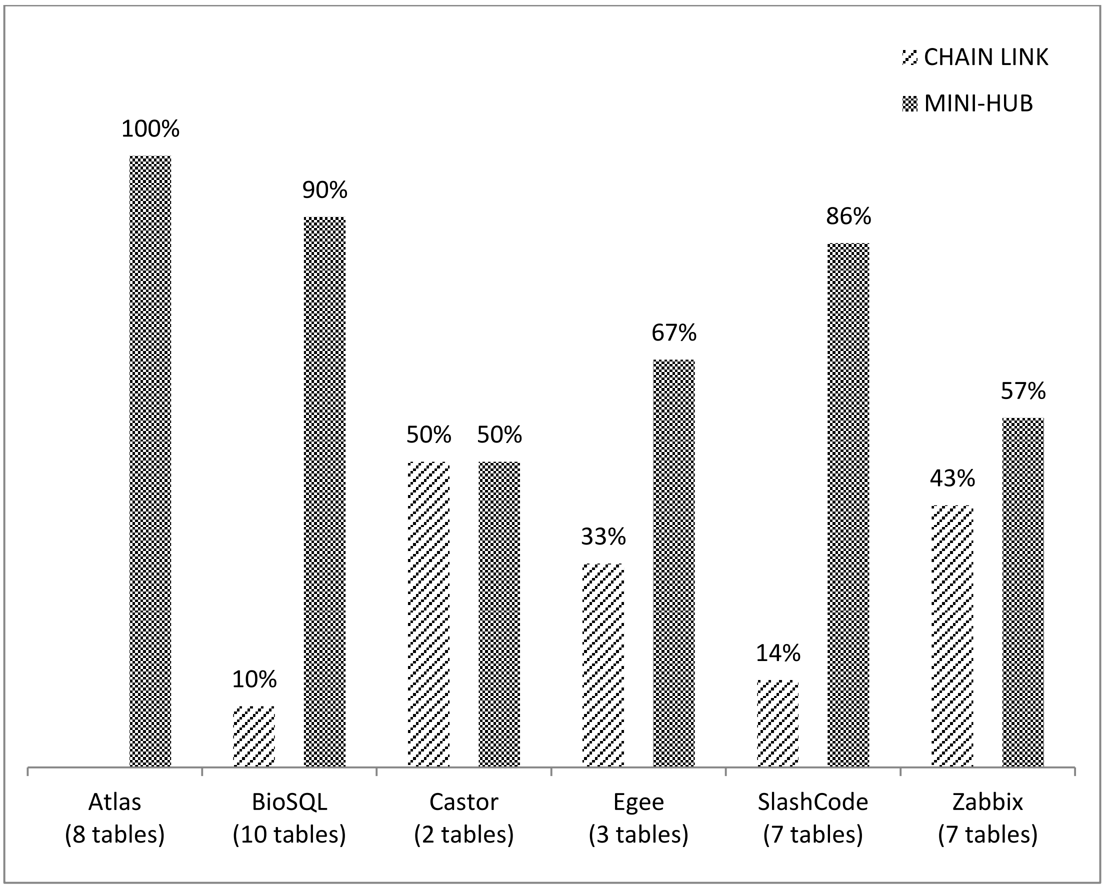


Figure 4.8 Breakdown of Tables over the Chain Link and Mini-Hub Categories

Figure 4.9 illustrates how the categories, previously described, are determined based on the topology of the Diachronic Graph.

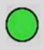
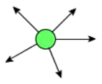
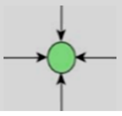
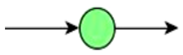
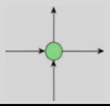
Name	Figure	Description
ISOLATED		Tables without edges
SOURCE		Tables with only outgoing edges
LOOKUP		Tables having only incoming edges
CHAIN LINK		Tables with exactly 1 incoming and exactly 1 outgoing edge
MINI-HUB		Tables with total degree >2 and both incoming and outgoing edges

Figure 4.9 Table Categories Based on the Topology of the Diachronic Graph

4.3.2 Rules for Table Classification

Having decided which the categories are, we are now ready to label the tables. Given a graph of any version of a schema's history, it is straightforward to assign labels of topological categories to every table due to the simplicity of the patterns. However, there exist tables that change label throughout their history (a phenomenon that we call change-of-category) and as a result we end up with the following categories of tables with respect to their labels:

- *Single label* tables, which have a unique topological label throughout their entire lives.
- *Multi-label* tables, which have more than one label during their existence in the dataset.

Figure 4.10 presents the distribution of tables between the ones with a single label and those with more than one label. Apart from Zabbix, in the rest of the datasets the majority of tables have a single label in their lives.

		#Tables with…	
Datasets	**Total #tables**	single label	>1 label
Atlas	88	76	12
BioSQL	45	39	6
Castor	91	84	7
Egee	12	9	3
SlashCode	126	97	29
Zabbix	58	30	28

Figure 4.10 Distribution of Tables over the Single and Multi-labels Categories

A problem that arises is that we would like to relate the labels of the tables to their activity profile and their survival potential and a multi-labeling scheme would not facilitate this attempt. To address this problem, we have manually inspected the tables with change-of-category and decided to assign a single label to each of them, since their number is so small that would not entail any major loss of information. We have distilled the phenomena of label changes for a table in the following list:

1. Changes that include an ephemeral transition to a different category and the return to the former category.
2. Changes from the *isolated* category to a different category.
3. Changes soon after the table's "birth".
4. Changes leading to labels assigned for a short period in terms of the number of versions.
5. Changes caused by the introduction or the removal of self-references to the table.

Figure 4.11 demonstrates the breakdown of multi-label tables according to the aforementioned enumeration of changes that induce label change. A subtle point to clarify is that the reported frequencies concern occurrences of label change and not of tables belonging to the respective category (i.e., a table can experience more than one label changes due to more than one types of changes). A second subtle point is that a single occurrence of a change may belong to more than one categories of the enumeration (for example, a change (a) from isolated to non-isolated, (b) soon after a table's birth pertains to both these two types of changes). We resolve this issue by counting only the occurrence in one of the two categories: the resolution of which category to assign to, is done with decreasing order over the enumerated list of the above enumeration (i.e., an occurrence is assigned to the first category to which it pertains).

	Type of Change					
Datasets	Ephemeral (DO-UNDO)	ISOLATED -> new category	Soon after birth	Short - lasting labels	Self-references	Other
Atlas	6	0	0	1	0	7
BioSQL	0	1	0	3	5	0
Castor	2	6	0	3	0	0
Egee	0	1	1	2	0	1
SlashCode	20	3	1	0	0	5
Zabbix	0	4	2	3	0	4

Figure 4.11 Occurrences of Label Changes per Type of Change

Having done all that, we discovered that the process of assigning a category label to multi-label tables can be automated by passing the history of labels of each table through a list of filters that either remove or ignore parts of the history with labels that would confuse the understanding of the true nature of the tables. The input in this automated process is the list of labels of a table's history, one label per version that the table exists. The history is then passed through the list of filters to remove the possibly bewildering parts and produce as an output a single label for the table.

Figure 4.12 summarizes the rules that represent the list of filters utilized to classify tables in the topological categories. We should clarify that this list of filters defines the order according to which the rules are applied on the list of labels of each table to produce a single label. If we had implemented the automatic process of filtering, we would have ended up with identical labels with those of the manual classification for the tables that abide by any of the rules R0-R5, but we would have misclassified few tables that eventually fire the rule R6.

At this point, we should define the terms *First Known Version* and *Most Frequent* category which are included in Figure 4.12. The *First Known Version* of a table refers to the first version that the table is present in the database's schema. The *Most Frequent* category for a table is the topological category with the highest frequency in table's life.

Figure 4.13 shows the misclassification rate of the automatic labeling process, in case the rule R6 was stricter allowing one instead of two categories. Except for the Atlas, all the datasets have the minimum misclassification rate when we use the most frequent category. *Observe that in the case of labeling via the most-frequent category, the range of misclassifications is between 0% and 3%, which we deem really low*. Although the misclassification rate in most datasets is not high, in the rest of our deliberations, *we adopt the labels derived from the manual classification process*, which provides a more accurate picture of tables' topological categories, taking into consideration the special features of the tables included.

Rule	**Description of Changes**	**Specific Criteria**	**Category Decision**
R0	No category change	-	The respective category
R1	Ephemeral category changes (DO-UNDO)	Changes must be successive	The first category prior to the first change
R2	Changing from ISOLATED to another category	-	The category after the change
R3	Changing category soon after the First Known Version (FKV)	The upper limit is set to 10 versions	The category after the change
R4	Changing to a category with short duration	Duration should not exceed 10 versions	The category prior to the change
R5	Changing category due to the presence of self-references	-	The category prior to the change
R6	Changes not abiding by any of the previous rules	-	The Most Frequent category or the category at the First Known Version (FKV)

Figure 4.12 Rules for Tables' Categories Determination

Datasets	#tables	Misclassified Tables (wrt to #tables)	
		Use Most Frequent Category	Use Category at FKV
Atlas	88	2%	0%
BioSQL	45	2%	2%
Castor	91	3%	7%
Egee	12	0%	17%
SlashCode	126	2%	5%
Zabbix	58	2%	16%

Figure 4.13 Misclassification Rate of Assigning Labels via the Automatic Process

In the remainder of this chapter we examine how tables' topological categories are related to various measures of their evolutionary behavior, such as their lives' duration, their survival potential, their update activity etc.

4.4 Relationship between Tables' Topological Categories and their Properties

Having determined the categories in the previous section, we are now capable of studying whether tables' topological categories are related with various measures of their evolutionary activity. Before that, we provide a general overview of how tables are classified in the topological categories after the classification process we performed in the six datasets.

Figure 4.14 depicts a heatmap with the breakdown of tables over the different categories defined in the forgoing section. The colors of the cells are based on their values creating a color scale that spans from white, soft red to intense red with the first indicating the lowest values, the second corresponding to values around the median and the last one highlighting the highest values. The groups with the highest cardinality, which are presented with intense red background color and white font, consist of *isolated* tables in 4 of the 6 datasets, in contrast to the two scientific datasets in which *source* tables form the most populated class.

Topological	**Datasets**					
Category	Atlas	BioSQL	Castor	Egee	SlashCode	Zabbix
ISOLATED	11	2	**75**	**6**	**35**	**22**
SOURCE	**38**	**29**	6	2	22	20
LOOKUP	32	8	9	1	7	11
MINI-HUB	6	6	1	0	4	2
CHAIN LINK	1	0	0	3	0	1
Total	**88**	**45**	**91**	**12**	**68**	**56**
Total w/o ISO	**77**	**43**	**16**	**6**	**33**	**34**

Figure 4.14 Breakdown of Tables over Topological Categories

Figure 4.15 depicts the distribution of the tables in the topological categories with respect to the total number of the tables.

Topological	**Datasets**					
Category	Atlas	BioSQL	Castor	Egee	SlashCode	Zabbix
ISOLATED	13%	4%	**82%**	**50%**	**51%**	**39%**
SOURCE	**43%**	**64%**	7%	17%	32%	36%
LOOKUP	36%	18%	10%	8%	10%	20%
MINI-HUB	7%	13%	1%	0%	6%	4%
CHAIN LINK	1%	0%	0%	25%	0%	2%
Total	**88**	**45**	**91**	**12**	**68**	**56**

Figure 4.15 Distribution of Tables over Categories including Isolated Category

We complement the absolute breakdown of tables with a breakdown of tables that have at least one inciting edge. Figure 4.16 shows how tables are spread over the categories after having removed tables of the *isolated* class. We highlight the maximum values with red color and bold style, the values that exceed the average by 10% with red color and those that are equal or lower than the average by 10% with blue color.

Topological	**Datasets**					
Category	Atlas	BioSQL	Castor	Egee	SlashCode	Zabbix
SOURCE	**49%**	**67%**	38%	33%	**67%**	**59%**
LOOKUP	42%	19%	**56%**	17%	21%	32%
MINI-HUB	8%	14%	6%	0%	12%	6%
CHAIN LINK	1%	0%	0%	**50%**	0%	3%
Total	**77**	**43**	**16**	**6**	**33**	**34**

Figure 4.16 Distribution of Tables over Categories excluding Isolated Category

The most interesting observations derived from the last figure can be summarized as follows:

- In 4 of the 6 datasets, the *source* tables constitute an overwhelming majority accounting for the 49% at least and 67% at most with respect to the total number of the tables with at least one edge.
- There appears a decreasing tendency for dependence, since the last two categories that represent complicated relationships include a small number of tables. In accordance with this tendency we see that in all datasets, except Egee, the *lookup* tables exceed the sum of *mini-hub* and *chain link* tables.
- The *chain link* category contains a negligible portion of tables that do not surpass the 3% of the total number of tables, except for the Egee dataset in which this category encompasses the one half of the tables. However, the small number of tables in the Egee dataset and the total absence of tables of this type in three other datasets are deterrent factors for preserving this class as an independent category. Thus, as we previously mentioned, it would be wiser to incorporate them in the *mini-hub* category forming a new category for which we will use the label *internal*.

Having presented the breakdown of values for the different topological categories of tables, we now move on to investigate whether the topological categories of tables are related to their evolutionary behavior. In the sequel, we will not include the Egee dataset in our study due to the small number of its tables and our intuition that any statistical results provided for this dataset would not offer a more adequate answer to the upcoming research questions.

4.4.1 Relationship between Topological Categories and Duration

First, we study how table duration is related to the topological categories. The research question that we attempt to address in this subsection can be stated as follows:

Research Question: is there a relationship between the topological category of a table and its duration?

The *duration* of a table represents the number of versions in which the table exists in the dataset. We decided to use the categories of duration presented in [VaZS15], where the authors define three different duration categories based on the measure of the normalized duration.

Terminology. The *normalized duration* of a table is defined as the number of versions that the table exists in the dataset over the total number of versions of its dataset.

Figure 4.17 presents the bounds of the duration categories as they derived from applying a k-means clustering based on the values of the normalized duration. The limits provided by k-means in [VaZS15] are 0.33 and 0.77, determining the following categories of tables:

i. Tables of short duration, which constitute the second most popular category with respect to the total number of the tables of the six datasets.

ii. Tables of medium duration.

iii. Tables of long duration, which account for more than half of the total number of tables included.

Tables…	**Range**	**#Tables**	**Percentages (wrt to the total #Tables)**
Short Lived	< 0.33	98	28%
Medium Lived	0.33-0.77	73	21%
Long Lived	> 0.77	179	51%
Total		**350**	**100%**

Figure 4.17 Distribution of Tables per Normalized Duration Category

Figure 4.18 depicts how tables in each dataset are spread over the categories of the normalized duration. We highlight with intense red color the dominant category, which in 4 of the 6 datasets is that of the *long lived* tables. The distribution of the tables over the duration categories among the different datasets can be summarized as follows:

- *Short lived* tables constitute a population that ranges from 23% to 32% of the total number of tables.
- *Medium lived* tables represent a population that varies from 14% to 28% with respect to the total number of tables, with the exception of BioSQL dataset in which tables with medium life duration represent the most populated category. This differentiation is mainly attributed to a significant schema restructuring at the middle of the database's life.
- *Long lived* tables add up to a population that ranges from 40% to 59% of the total number of tables, with the exception of BioSQL.

BREAKDOWN OF TABLES WRT NORMALIZED DURATION (PERCENTAGES OVER TOTAL #TABLES)

	NORMALIZED DURATION CATEGORY			
	SHORT LIVED	MEDIUM LIVED	LONG LIVED	Total #Tables
Atlas	32%	14%	**55%**	88
BioSQL	31%	**38%**	31%	45
Castor	24%	16%	**59%**	91
SlashCode	23%	19%	**58%**	69
Zabbix	32%	28%	**40%**	57

Figure 4.18 Distribution of Tables over the Normalized Duration Categories

Figure 4.19 illustrates the distribution of the tables with respect to the combination of their topological and duration categories. In the upper part of the figure the tables of the *isolated* category are included, while the lower part ignores them and computes the respective percentages over the total number of the tables with at least one reference. The most interesting observations derived from the data shown in Figure 4.19 are outlined as follows:

- The most populated category in three of the five datasets is that of the *source* tables with *medium* or *long* durations.
- The least populated categories, in all datasets apart from BioSQL, are those of the *internal* tables with *short* or *medium* life durations.

Figure 4.20 presents the distribution of the tables over the topological and duration categories within each of the topological categories. We see that the distributions of the *source* tables are in accordance with the aggregate ones in three of the five datasets, except for the Castor and the SlashCode datasets. It is also obvious that in all datasets *lookup* tables with long life duration exceed the respective aggregate percentages. To put in a nutshell the most significant commonalities among the datasets, we mention the following observations:

- The majority of *lookup* tables tend to live *long* lives in all the datasets.
- The *long* lived category is also the most popular in case of the *source* tables in 4 out of the 5 datasets, with the exception of the BioSQL dataset.
- The *internal* tables avoid lives of *short* or *medium* duration, except for those of the BioSQL and the Zabbix datasets, even though they do not form a population that exceeds the 10% of the total number of their dataset's tables. In case of BioSQL, we attribute the different behavior to the major schema restructuring occurred at the middle of the database's life while the *short* lives of the *internal* tables of Zabbix are due to occasional deletions.
- In contrast to the previous topological categories, the *isolated* tables incline to lives of *short* and *medium* duration, apart from those of the Castor dataset that demonstrate a clear proclivity for lives of *long* duration.

Having quantified the number of tables per topological and duration categories, we performed the Chi-square and Fisher tests to assert whether tables' behavior concerning their normalized duration is differentiated due to their topological categories. The contingency table we used consists of four rows, each representing a topological category, and three columns that correspond to the three duration categories (short, medium and long lived). Both tests cannot strongly support that the differences among the duration categories are caused by the topology of the tables, since the p-values that do not exceed the limit of 5% are 4.998E-06 in case of the Atlas dataset and 3.349E-02 for SlashCode.

To sum up, we studied how tables are spread over the combination of their topological and duration categories identifying several duration-related patterns, out of which we distinguish internal and lookup tables' tendency to lives of long duration and the isolated tables' disinclination to longevity. However, the statistical evidence does not allow us to emphatically suggest that there is a correlation between tables' topological categories and their duration.

BREAKDOWN OF ALL TABLES PER TOPOLOGICAL AND DURATION CATEGORIES (PERCENTAGES OVER TOTAL #TABLES)

TOPOLOGICAL CATEGORY

		ISOLATED			**SOURCE**			**LOOKUP**			**INTERNAL**			**Aggregate per Duration Category**		
	Total #Tables	SHORT LIVED	MEDIUM LIVED	LONG LIVED	SHORT LIVED	MEDIUM LIVED	LONG LIVED	SHORT LIVED	MEDIUM LIVED	LONG LIVED	SHORT LIVED	MEDIUM LIVED	LONG LIVED	SHORT LIVED	MEDIUM LIVED	LONG LIVED
Atlas	88	3%	8%	1%	17%	5%	22%	11%	1%	24%	0%	0%	8%	32%	14%	55%
BioSQL	45	2%	2%	0%	22%	24%	18%	4%	4%	9%	2%	7%	4%	31%	38%	31%
Castor	91	24%	13%	45%	0%	1%	5%	0%	2%	8%	0%	0%	1%	24%	16%	59%
SlashCode	69	19%	12%	20%	3%	6%	23%	0%	1%	9%	0%	0%	6%	23%	19%	58%
Zabbix	57	14%	16%	9%	11%	7%	18%	4%	5%	11%	2%	0%	4%	32%	28%	40%

BREAKDOWN OF TABLES WITH AT LEAST ONE EDGE PER TOPOLOGICAL AND DURATION CATEGORIES (PERCENTAGES OVER TOTAL #TABLES)

TOPOLOGICAL CATEGORY

		SOURCE			**LOOKUP**			**INTERNAL**			**Aggregate per Duration Category**		
	Total #Tables	SHORT LIVED	MEDIUM LIVED	LONG LIVED	SHORT LIVED	MEDIUM LIVED	LONG LIVED	SHORT LIVED	MEDIUM LIVED	LONG LIVED	SHORT LIVED	MEDIUM LIVED	LONG LIVED
Atlas	77	19%	5%	25%	13%	1%	27%	0%	0%	9%	32%	6%	61%
BioSQL	43	23%	26%	19%	5%	5%	9%	2%	7%	5%	30%	37%	33%
Castor	16	0%	6%	31%	0%	13%	44%	0%	0%	6%	0%	19%	81%
SlashCode	33	6%	12%	48%	0%	3%	18%	0%	0%	12%	6%	15%	79%
Zabbix	34	18%	12%	29%	6%	9%	18%	3%	0%	6%	26%	21%	53%

Figure 4.19 Distribution of Tables per Topological and Duration Categories with and without the ISOLATED Category

PROBABILITY FOR A TABLE OF A TOPOLOGICAL CATEGORY TO BELONG TO A CERTAIN DURATION CATEGORY (PERCENTAGES OVER TOTAL #TABLES OF EACH TOPOLOGICAL CATEGORY)

	TOPOLOGICAL CATEGORY																			
	ISOLATED				**SOURCE**				**LOOKUP**				**INTERNAL**				**Aggregate per Duration Category**			
	Total #Tables	SHORT LIVED	MEDIUM LIVED	LONG LIVED	Total #Tables	SHORT LIVED	MEDIUM LIVED	LONG LIVED	Total #Tables	SHORT LIVED	MEDIUM LIVED	LONG LIVED	Total #Tables	SHORT LIVED	MEDIUM LIVED	LONG LIVED	Total #Tables	SHORT LIVED	MEDIUM LIVED	LONG LIVED
Atlas	11	27%	**64%**	**9%**	38	39%	**11%**	**50%**	32	31%	**3%**	**66%**	7	0%	0%	**100%**	88	32%	14%	55%
BioSQL	2	**50%**	**50%**	0%	29	34%	**38%**	**28%**	8	**25%**	**25%**	**50%**	6	**17%**	**50%**	33%	45	31%	38%	31%
Castor	75	29%	**16%**	**55%**	6	0%	**17%**	**83%**	9	0%	**22%**	**78%**	1	0%	0%	**100%**	91	24%	16%	59%
SlashCode	35	37%	**23%**	**40%**	22	**9%**	18%	**73%**	7	0%	**14%**	**86%**	4	0%	0%	**100%**	68	22%	19%	59%
Zabbix	22	36%	**41%**	**23%**	20	30%	**20%**	**50%**	11	**18%**	27%	**55%**	3	**33%**	0%	**67%**	56	30%	29%	41%

Figure 4.20 Probability for a Table of a Topological Category to Belong to a Certain Duration Category

4.4.2 Relationship between Topological Categories and Survival

The next property that we study in reference to the topological categories is the tables' survival potential. We describe a table as a "survivor" if the table exists in the last known version of its dataset. The respective research question that we attempt to address is the following:

Research Question: is there a relationship between the topological category of a table and its survival potential?

Figure 4.21 depicts the population of survivors in each dataset with respect to the topological categories they belong to. The including percentages are computed with reference to the total number of each dataset's tables. The red and blue colors represent the most and the least populated categories, respectively.

DISTRIBUTION OF SURVIVORS WITH AT LEAST ONE EDGE PER TOPOLOGICAL CATEGORY (PERCENTAGES OVER TOTAL #TABLES)

		TOPOLOGICAL CATEGORY (FOR SURVIVORS)				**Aggregate per Topological Category (ind. of survival)**		
	Total #Tables	**SOURCE**	**LOOKUP**	**INTERNAL**	Aggregate %Survivors	**SOURCE**	**LOOKUP**	**INTERNAL**
Atlas	77	**39%**	34%	**9%**	82%	49%	42%	9%
BioSQL	43	**44%**	12%	**9%**	65%	67%	19%	14%
Castor	16	31%	**44%**	**6%**	81%	38%	56%	6%
SlashCode	33	**64%**	21%	**12%**	97%	67%	21%	12%
Zabbix	34	**53%**	24%	**6%**	85%	59%	32%	9%

Figure 4.21 Distribution of "Survivors" per Topological Category

It is obvious that, in four out of the five datasets studied, there appears a decreasing sequence of percentages of the tables included among the categories as presented in Figure 4.21, with the highest cardinality of survivors to be assigned to the *source* tables and the lowest one attributed to the *internal* tables. Figure 4.21 also contains the aggregate percentages of the survivors, which are surprisingly high in all datasets varying from 65% to 97% of the corresponding total number of tables. The last three columns include the overall percentages of tables per topological category, regardless of their survival potential. We see that the "survivors" of the *internal* category follow the respective aggregate percentages, which means that the survival potential for these tables will be high.

In the previous figure we ignored the existence of the *isolated* tables, counting the "survivors" with reference to the tables with at least one edge. If we include the *isolated* tables, we will observe few differentiations concerning the spread of the tables among the topological categories. Figure 4.22 depicts the distribution of the tables-survivors over the topological categories including the *isolated* category. Once again, the red color signifies the most populated category, in terms of the number of survivors, and the blue color the least one. We should mention that in two datasets, namely Castor and SlashCode, the *isolated* "survivors" form a clear majority, which is largely explained by the strong presence of the tables of the *isolated* group in these two datasets. The *isolated* "survivors" are the second most populated group of tables in Zabbix, as opposed to Atlas where they are the second least popular category. Finally, BioSQL does not encompass "survivors" of the *isolated* category at all.

DISTRIBUTION OF ALL SURVIVORS PER TOPOLOGICAL CATEGORY (PERCENTAGES OVER TOTAL #TABLES)

		TOPOLOGICAL CATEGORY (FOR SURVIVORS)					**Aggregate per Topological Category (ind. of survival)**			
	Total #Tables	**ISOLATED**	**SOURCE**	**LOOKUP**	**INTERNAL**	Aggregate %Survivors	**ISOLATED**	**SOURCE**	**LOOKUP**	**INTERNAL**
Atlas	88	11%	**34%**	30%	**8%**	83%	13%	43%	36%	8%
BioSQL	45	**0%**	**42%**	11%	9%	62%	4%	64%	18%	13%
Castor	91	**67%**	5%	8%	**1%**	81%	82%	7%	10%	1%
SlashCode	68	**44%**	31%	10%	**6%**	91%	51%	32%	10%	6%
Zabbix	56	30%	**32%**	14%	**5%**	82%	39%	36%	20%	5%

Figure 4.22 Distribution of "Survivors" per Topological Category (including ISOLATED)

Figure 4.23 illustrates how the tables that survive are spread over the topological categories with respect to the total number of tables of each category. The patterns that we observe in this figure can be outlined as follows:

- The distributions of the survivors of the categories *source* and *lookup* are similar to the respective aggregate distributions in all datasets, with the exception of the *lookup* survivors of the Zabbix dataset.
- The *internal* category ensures that each participating table is sure to survive and this observation holds in all the datasets apart from BioSQL. In all the datasets, the percentages of the *internal* survivors exceed the respective aggregate portions of survivors.

PROBABILITY OF SURVIVAL PER TOPOLOGICAL CATEGORY (PERCENTAGES OVER TOTAL #TABLES OF EACH TOPOLOGICAL CATEGORY)

	TOPOLOGICAL CATEGORY									**Aggregate Survival Probability**	
	SOURCE			**LOOKUP**			**INTERNAL**				
	Total #Tables	#Survivors	%Survivors	Total #Tables	#Survivors	%Survivors	Total #Tables	#Survivors	%Survivors	Total #Tables	%Survivors
Atlas	38	30	**79%**	32	26	81%	7	7	**100%**	77	82%
BioSQL	29	19	66%	8	5	**63%**	6	4	**67%**	43	65%
Castor	6	5	83%	9	7	**78%**	1	1	**100%**	16	81%
SlashCode	22	21	**95%**	7	7	**100%**	4	4	**100%**	33	97%
Zabbix	20	18	90%	11	8	**73%**	3	3	**100%**	34	85%

Figure 4.23 Probability of Survival per Topological Category

Figure 4.24 shows the probability of survival for the *isolated* tables. The survival potential for the tables of this category is significantly high in all the datasets except for BioSQL. It is also noteworthy that the including percentages approach the aggregate ones in four of the five datasets.

PROBABILITY OF SURVIVAL FOR THE ISOLATED TABLES (PERCENTAGES OVER TOTAL #TABLES)

				Aggregate Survival Probability	
	Total #Tables	#Survivors	%Survivors	Total #Tables	%Survivors
Atlas	11	10	91%	88	83%
BioSQL	2	0	0%	45	62%
Castor	75	61	81%	91	81%
SlashCode	35	30	86%	68	91%
Zabbix	22	17	85%	56	86%

Figure 4.24 Probability of Survival for the ISOLATED Tables

The high percentages of "survivors", regardless of the topological categories, prejudiced us against the impact of the categories on the survival potential of a table. This intuition was confirmed by the statistical tests we conducted by forming 4x2 contingency tables, with their rows corresponding to the topological categories (isolated, source, lookup, internal) and their two columns representing the populations of the tables that exist in the last known version and those that do not. The lowest p-value the Chi-square and Fisher tests returned was 0.3238, indicating that there are no sufficient data to support the correlation between the topological categories and the survival potential.

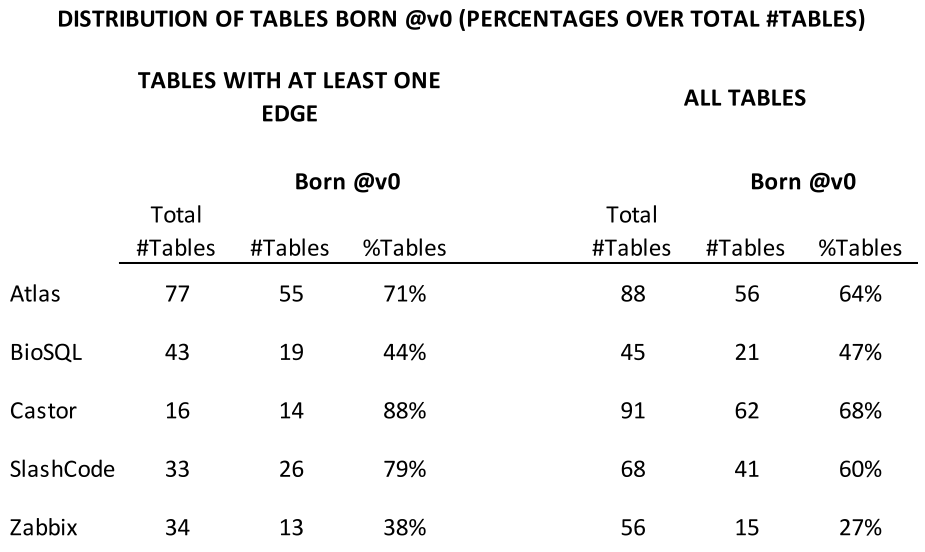
Overall, we should stress the high survival potential of the tables disregarding their topological categories, which along with the statistical results are strong indications that tables' topology is not likely to be related to their probability to exist in the last known version.

4.4.3 Relationship between Tables' Topological Categories and Birth Version

In this subsection we investigate if birth versions of the tables are related to their topological categories. We are particularly interested in the relationship between the probability that a table is born in the originating version of the schema history and the topological category it belongs to. In this context, we can formulate the relevant research question as follows:

Research Question: how is the topological category of a table related to the probability of being born in the originating version of its dataset's schema history?

Figure 4.25 illustrates the populations of the tables born in the very first version of their datasets history. The left part of the figure ignores tables of the *isolated* category, while the right part includes them. We can observe that in three out of the five datasets, the tables born in the originating version form overwhelming majorities that exceed the 70% of the total number of the tables.



DISTRIBUTION OF TABLES BORN @v0 (PERCENTAGES OVER TOTAL #TABLES)

	TABLES WITH AT LEAST ONE EDGE			**ALL TABLES**		
		Born @v0			**Born @v0**	
	Total #Tables	#Tables	%Tables	Total #Tables	#Tables	%Tables
Atlas	77	55	71%	88	56	64%
BioSQL	43	19	44%	45	21	47%
Castor	16	14	88%	91	62	68%
SlashCode	33	26	79%	68	41	60%
Zabbix	34	13	38%	56	15	27%

Figure 4.25 Populations of Tables (left: without ISOLATED; right: with ISOLATED) Born in the Originating Version

In Figure 4.26 we present how the tables born in the first version are spread over the topological categories. The red and blue colors indicate the most and the least populated topological categories with respect to the total number of each dataset's tables.

It is worth mentioning that, in three of the five datasets, *source* tables born in the first version of their dataset's history are the most popular category, even though only in one of them, namely BioSQL, they are the dominating group as it is illustrated in Figure 4.15 that presents the distribution of the tables over the topological categories. In Atlas, we notice that *lookup* tables born in the very first version exceed those of the *source* category, though the latter are the most popular among the dataset's tables irrespectively of their "birth" version.

DISTRIBUTION OF TABLES BORN @v0 PER TOPOLOGICAL CATEGORY
(PERCENTAGES OVER TOTAL #TABLES)

		TOPOLOGICAL CATEGORY				
	Total #Tables	**ISOLATED**	**SOURCE**	**LOOKUP**	**INTERNAL**	**Total**
Atlas	88	**1%**	26%	**28%**	8%	64%
BioSQL	45	**4%**	**24%**	9%	9%	47%
Castor	91	**53%**	5%	9%	**1%**	68%
SlashCode	68	22%	**24%**	9%	**6%**	60%
Zabbix	56	**4%**	**11%**	9%	**4%**	27%

Figure 4.26 Distribution of Tables Born in the Originating Version per Topological Category

Figure 4.27 depicts the potential the tables of each topological category have to exist in the first version of their schema's history.

The commonalities that we encounter with reference to the probability of tables being "born" in the earliest version of their schema can be summarized as follows:

- The tables of the *internal* category are 100% certain to be "born" in the originating version in three out of the five datasets. In BioSQL and Zabbix, although the overall population of the *internal* tables is not present in the first version, the corresponding percentages are high (67% in both cases).
- *Lookup* tables have higher probabilities to be "born" in the first version compared to the respective average probability, and in fact, their majority is present at the first version for four out of five datasets. The same holds for the corresponding probabilities of the *source* tables.

PROBABILITY TO BE BORN @v0 PER TOPOLOGICAL CATEGORY (PERCENTAGES OVER TOTAL #TABLES OF EACH TOPOLOGICAL CATEGORY)

	TOPOLOGICAL CATEGORY								**AGGREGATE BORN @v0**			
	ISOLATED		**SOURCE**		**LOOKUP**		**INTERNAL**		**TABLES WITH AT LEAST ONE EDGE**		**ALL TABLES**	
	Total #Tables	Born @v0	Total #Tables	Born @v0	Total #Tables	Born @v0	Total #Tables	Born @v0	Total #Tables	Born @v0	Total #Tables	Born @v0
Atlas	11	**9%**	38	61%	32	78%	7	**100%**	77	71%	88	64%
BioSQL	2	**100%**	29	**38%**	8	50%	6	67%	43	44%	45	47%
Castor	75	**64%**	6	83%	9	89%	1	**100%**	16	88%	91	68%
SlashCode	35	**43%**	22	73%	7	86%	4	**100%**	33	79%	68	60%
Zabbix	22	**9%**	20	30%	11	45%	3	**67%**	34	38%	56	27%

Figure 4.27 Probability to be “born” in the First Version per Topological Category

- The tables of the *isolated* category have the lowest potential for being "born" in the originating version of their datasets, in four of the five datasets. Equivalently, we can claim that it is easier to add tables of this category over the course of a database's schema evolution than introducing *lookup* or *internal* tables.
- The probability for a *source* table to be introduced in the first version of its dataset's history is, approximately, in accordance with the average probability and, in all datasets, is lower than the respective potential of the *lookup* tables.

The common features among the datasets related to the probability for a table to be "born" in the originating version if it belongs to a certain topological category are supported to some extent by the statistical evidence that assess the independence of the birth version from the topological categories. Specifically, we performed the Chi-square and Fisher statistical tests by utilizing a contingency table consisted of four rows representing the topological categories and two columns corresponding to tables born in the first version and those that are not. The p-values that do not exceed the limit of 5% are 4.74E-02 for Atlas, 1.36E-02 for SlashCode and 3.22E-02 for Zabbix.

To sum up, we observed that internal and lookup tables are more likely to be "born" in the originating version of their dataset's history, which, expressed in a different way, means that it is quite unlikely that they are "born" after this version. In contrast, isolated and source tables are less probable to be introduced in the first version, which entails that it is more probable that versions succeeding the originating one include new tables of these two categories. The behavior of the lookup and the internal tables can be attributed to the so-called gravitation to rigidity pattern [VaZS17], according to which it is fairly improbable that dependency-magnet tables, as those of the two aforementioned categories, experience any kind of change in later versions of database's schema. In this context, we can assume that administrators prefer creating tables that attract foreign keys in the early if not in the originating versions of the database in order to avoid changes caused by inserting them in subsequent versions.

4.4.4 Relationship between Tables' Topological Categories and Update Activity

The next issue that we are interested in is that of the update profile of the tables with respect to their topological categories. Thus, the research question that arises can be put in the following way:

Research Question: is there a relationship between the topological category of a table and its update activity?

To ease the process of analyzing tables' update behavior with respect to their topological categories we decided to utilize the activity classes defined in [VaZS15], which are summarized as follows:

i. *Rigid* tables, which experience no updates throughout their entire lives in their datasets.

ii. *Quiet* tables, with the total number of updates not exceeding the value of 5 and the Average Transitional Update (ATU) to be less than 0.1.

iii. *Active* tables, which undergo more than 5 updates and have an ATU higher than 0.1.

Terminology. The *Average Transitional Update* (ATU) of a table is defined as the fraction of the sum of updates the table undergoes throughout its life over its duration. [VaZS15]

Figure 4.28 presents the distribution of the tables over the aforementioned activity classes. The upper part of the figure ignores the presence of the *isolated* tables, whereas the lower part includes them. The largest and the smallest classes in terms of the tables' population are highlighted with red and blue colors, respectively.

Ignoring the *isolated* tables, we observe that, in four of the five datasets, the most multitudinous group is that of the *quiet* tables, accounting for nearly or more than the one half of tables' population. But, if we take into account the *isolated* tables, we can identify a decrease of small or large magnitude in the numbers of *quiet* tables in all the datasets, apart from Atlas, and a simultaneous increase in the cardinality of the *rigid* tables. As for the *active* tables, if we include the *isolated* category, there appears a decrease in their numbers in all the datasets to an extent varying from 1% to 11% with respect to the total number of the tables.

BREAKDOWN OF TABLES WITH AT LEAST ONE EDGE WRT ACTIVITY CLASS (PERCENTAGES OVER TOTAL #TABLES)

	Total	Activity Class			Activity Class (%)		
	#Tables	RIGID	QUIET	ACTIVE	RIGID	QUIET	ACTIVE
Atlas	77	15	37	25	**19%**	**48%**	32%
BioSQL	43	14	13	16	33%	**30%**	**37%**
Castor	16	7	7	2	**44%**	**44%**	**13%**
SlashCode	33	3	19	11	**9%**	**58%**	33%
Zabbix	34	11	21	2	32%	**62%**	**6%**

BREAKDOWN OF ALL TABLES WRT ACTIVITY CLASS (PERCENTAGES OVER TOTAL #TABLES)

	Total	Activity Class			Activity Class (%)		
	#Tables	RIGID	QUIET	ACTIVE	RIGID	QUIET	ACTIVE
Atlas	88	18	43	27	**20%**	**49%**	31%
BioSQL	45	16	13	16	**36%**	**29%**	**36%**
Castor	91	57	31	3	**63%**	34%	**3%**
SlashCode	68	15	38	15	**22%**	**56%**	**22%**
Zabbix	56	23	30	3	41%	**54%**	**5%**

Figure 4.28 Distribution of Tables per Activity Class (top: without the ISOLATED; bottom: with the ISOLATED)

Next, we examine the impact of the topological categories on tables' update activity. Figure 4.29 shows how tables are divided into the different combinations of the topological and activity categories. As we mentioned before, we used the red color to signify the most populated group of tables in each dataset and the blue one for the least popular group after the groups with no including tables. As far as the distribution of the tables of the different topological categories over the activity classes is concerned, we should mention the following observations:

- In two of the five datasets, namely Atlas and Zabbix, the *source* tables with moderate update activity are the most popular with respect to the total number of the tables. In Castor and SlashCode, the *isolated* tables with no and quiet update activity, respectively, form the leading groups of tables, while in BioSQL we see that the most popular groups are those of the *source* tables with all kinds of update activities.

- We observe that in the least populated groups are included the *internal* tables with moderate activity in BioSQL, Castor and SlashCode, the *isolated* tables with intense activity in Zabbix and the *lookup* tables with no updates in Atlas.

The upper part of Figure 4.30 depicts the probability for a table of a certain topological category to develop a certain update activity during its existence in its dataset. Once again, the red and blue colors correspond to the largest and smallest groups respectively, but in this case with reference to the number of tables of each topological category.

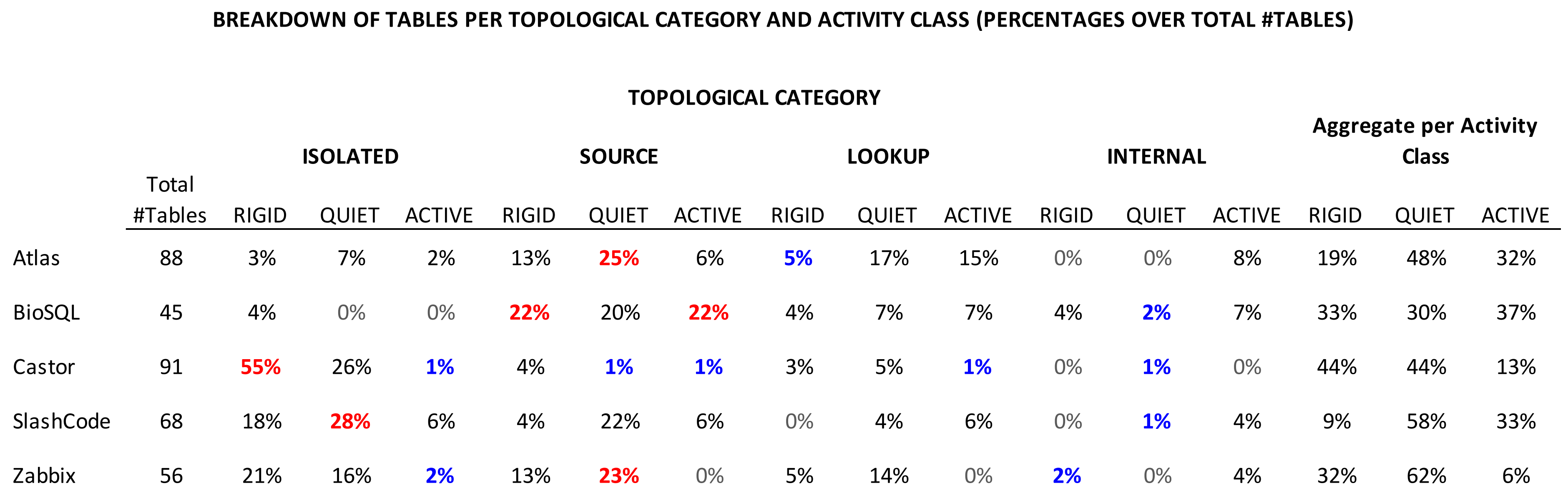
We outline the most interesting information derived from this figure in the following list:

- *Isolated* tables experience no or few updates with a probability that is higher than 82%.

- The likelihood for a *source* table to undergo no or few changes throughout its life is at least 82% in all datasets, apart from BioSQL.

- In three of the five datasets, the *lookup* tables with intense update activity exceed 38%, while those of the Castor and Zabbix datasets are pertained to quiet lives in terms of the changes they experience.

- In four of the five datasets, the *internal* tables are expected to undergo numerous updates.



BREAKDOWN OF TABLES PER TOPOLOGICAL CATEGORY AND ACTIVITY CLASS (PERCENTAGES OVER TOTAL #TABLES)

TOPOLOGICAL CATEGORY

	Total #Tables	ISOLATED			SOURCE			LOOKUP			INTERNAL			Aggregate per Activity Class		
		RIGID	QUIET	ACTIVE	RIGID	QUIET	ACTIVE	RIGID	QUIET	ACTIVE	RIGID	QUIET	ACTIVE	RIGID	QUIET	ACTIVE
Atlas	88	3%	7%	2%	13%	**25%**	6%	**5%**	17%	15%	0%	0%	8%	19%	48%	32%
BioSQL	45	4%	0%	0%	**22%**	20%	**22%**	4%	7%	7%	4%	**2%**	7%	33%	30%	37%
Castor	91	**55%**	26%	**1%**	4%	**1%**	**1%**	3%	5%	**1%**	0%	**1%**	0%	44%	44%	13%
SlashCode	68	18%	**28%**	6%	4%	22%	6%	0%	4%	6%	0%	**1%**	4%	9%	58%	33%
Zabbix	56	21%	16%	**2%**	13%	**23%**	0%	5%	14%	0%	**2%**	0%	4%	32%	62%	6%

Figure 4.29 Distribution of Tables per Topological and Activity Categories

PROBABILITY FOR A TABLE OF A TOPOLOGICAL CATEGORY TO DEVELOP A CERTAIN UPDATE ACTIVITY (PERCENTAGES OVER TOTAL #TABLES OF EACH TOPOLOGICAL CATEGORY)

TOPOLOGICAL CATEGORY

	ISOLATED				**SOURCE**				**LOOKUP**				**INTERNAL**				**Aggregate per Activity Class**			
	Total #Tables	RIGID	QUIET	ACTIVE	Total #Tables	RIGID	QUIET	ACTIVE	Total #Tables	RIGID	QUIET	ACTIVE	Total #Tables	RIGID	QUIET	ACTIVE	Total #Tables	RIGID	QUIET	ACTIVE
Atlas	11	27%	**55%**	**18%**	38	29%	**58%**	**13%**	32	**13%**	**47%**	41%	7	0%	0%	**100%**	88	20%	49%	31%
BioSQL	2	**100%**	**0%**	**0%**	29	**34%**	**31%**	**34%**	8	**25%**	**38%**	**38%**	6	33%	**17%**	**50%**	45	36%	29%	36%
Castor	75	**67%**	32%	**1%**	6	**67%**	**17%**	**17%**	9	33%	**56%**	**11%**	1	0%	**100%**	0%	91	63%	34%	3%
SlashCode	35	34%	**54%**	**11%**	22	**14%**	**68%**	18%	7	0%	**43%**	**57%**	4	0%	**25%**	**75%**	68	22%	56%	22%
Zabbix	22	**55%**	41%	**5%**	20	**35%**	**65%**	0%	11	**27%**	**73%**	0%	3	**33%**	0%	**67%**	56	41%	54%	5%

PROBABILITY FOR A TABLE OF AN ACTIVITY CLASS TO BELONG TO A CERTAIN TOPOLOGICAL CATEGORY (PERCENTAGES OVER TOTAL #TABLES OF EACH ACTIVITY CLASS)

ACTIVITY CLASS

	RIGID					**QUIET**					**ACTIVE**					**Aggregate per Topological Category**				
	Total #Tables	ISOLATED	SOURCE	LOOKUP	INTERNAL	Total #Tables	ISOLATED	SOURCE	LOOKUP	INTERNAL	Total #Tables	ISOLATED	SOURCE	LOOKUP	INTERNAL	Total #Tables	ISOLATED	SOURCE	LOOKUP	INTERNAL
Atlas	18	**17%**	**61%**	22%	0%	43	**14%**	**51%**	35%	0%	27	**7%**	19%	**48%**	26%	88	13%	43%	36%	8%
BioSQL	16	**13%**	**63%**	**13%**	**13%**	13	0%	**69%**	23%	**8%**	16	0%	**63%**	**19%**	**19%**	45	4%	64%	18%	13%
Castor	57	**88%**	7%	**5%**	0%	31	**77%**	**3%**	16%	**3%**	3	**33%**	**33%**	**33%**	0%	91	82%	7%	10%	1%
SlashCode	15	**80%**	**20%**	0%	0%	38	**50%**	39%	8%	**3%**	15	**27%**	**27%**	**27%**	**20%**	68	51%	32%	10%	6%
Zabbix	23	**52%**	30%	13%	**4%**	30	30%	**43%**	**27%**	0%	3	**33%**	0%	0%	**67%**	56	39%	36%	20%	5%

Figure 4.30 Probability for a Table of a Topological Category to Develop Specific Update Activity and vice versa

The bottom part of Figure 4.30 presents the probability for a table with a certain activity profile to belong to a specific topological category. In a nutshell, we can identify the subsequent commonalities among the datasets:

- The likelihood for a *rigid* table to be *source* is very high, especially in the datasets with no strong presence of *isolated* tables, while in datasets with numerous *isolated* tables, the *rigid* tables are more likely to be *isolated*. On the other hand, it is not quite possible for a *rigid* table to be *lookup*, since in all datasets this probability is less than the average one, and it is completely impossible a *rigid* table to be *internal* in three of the five datasets.

- In three of the five datasets, *quiet* tables are likely to belong to the *source* category, with the exceptions of Castor and SlashCode, in which *quiet* tables tend to be *isolated*. It is also obvious that the distribution of the *quiet* tables over the topological categories is in agreement with the aggregate one in all datasets.

- As for the *active* tables we notice a tendency towards categories of high topological complexity. This is verified by the fact that, in all datasets, the chances for an *active* table to belong to a topologically complex category are higher compared to the average probabilities. This is another way to identify *internals*' inclination towards intense update activity.

The statistical evidence provided by Chi-square and Fisher tests is fairly strong. For each dataset, we utilized a contingency table consisted of four rows, each of which represents a topological category and three columns corresponding to the different activity classes. The p-values derived from these tests are below the critical value of 5% in four of the five datasets, ranging from 9.6E-05 (Zabbix) to 3.89E-02 (Castor). *The statistical results confirm that tables with different topological categories are subjects to different amounts of updates.*

Altogether, we established that the topological category of a table is related to its update activity. Giving a summary of the findings, we can associate isolated and source tables with no or few updates, lookup tables with few or many changes and internal tables with many updates. These two different patterns can be regarded as an example of the "electrolysis" pattern presented in [VaZa17], where the authors identified two completely inverse behaviors concerning the relationship between tables' duration and their survival potential, with "dead" tables living for short durations and "survivors" related to lives of long duration. In the same sense, we can claim that topologically simplest tables are associated with few or no changes, whereas complex tables in terms of their topology are related to lives of intense update activity.

4.4.5 Relationship between Tables' Topological Categories and Size Change

Studying the relationship between tables' topological categories and their activity profiles we were surprised by the significant portions of *lookup* tables that undergo few or many updates over their lives in three of the five datasets. One would expect that tables which are dependency magnets are not prone to changes, since the dependents are certain to be affected. Given that, we decided to study how the topological category of a table is related to its size change between its first and last known versions. Naturally, the relative research question is expressed in the following way:

Research Question: how is the topological category of a table related to its size change?

Intuitively, we classified tables with respect to the scale of their size change in three categories that each of them expresses size reduction, stability or expansion. The scale of one table's size change is defined as the fraction of its size in the last version over its size in its first version. In a nutshell, the three size scale categories can be defined as follows:

i. *Scale down*, when there is a reduction in table's size, with the respective size scale to be less than 1.

ii. *Steady*, when table's sizes in the first and last versions are even, with the scale to be equal to 1.

iii. *Scale up*, when there is an expansion in table's size, with the scale to be greater than 1.

Figure 4.31 shows how the tables of each dataset are spread over the size scale categories. It is obvious that in all the datasets more than one half of the tables remain steady in terms of their size, while a considerable number of tables expand their size between their first and last versions. As for those that downsize their number of attributes, we observe that they do not constitute groups that exceed the 10% of the total number of the tables.

BREAKDOWN OF TABLES PER SIZE SCALE CATEGORY (PERCENTAGES OVER TOTAL #TABLES)

	Total #tables	Size Scale Categories <=0,99	1	>1
Atlas	88	6%	69%	25%
BioSQL	45	7%	53%	40%
Castor	91	3%	67%	30%
SlashCode	68	3%	50%	47%
Zabbix	56	2%	55%	43%

Figure 4.31 Distribution of Tables per Size Scale Category

We present in the upper part of Figure 4.32 the distribution of the tables over their size scale and topological category. The percentages included are quantified with reference to the total number of tables of each dataset. The red and blue colors represent the largest and smallest groups of tables with respect to the total number of tables, without taking into account the categories with no participating tables.

We outline the most noteworthy information derived from the upper part of Figure 4.32 in the upcoming list:

- In three out of the five datasets, the largest group of tables is that consisted of *isolated* tables with *steady* size scale. In Atlas and BioSQL the most populated category comprises the *source* tables with *steady* size scale.
- We should also mention the low percentages of tables that experience a size reduction in all topological categories. The corresponding values do not surpass the 2% of the total number of tables of each dataset.

BREAKDOWN OF TABLES PER TOPOLOGICAL AND SIZE SCALE CATEGORIES (PERCENTAGES OVER TOTAL #TABLES)

TOPOLOGICAL CATEGORY

		ISOLATED			**SOURCE**			**LOOKUP**			**INTERNAL**			**Aggregate per Size Scale Category**		
	Total #Tables	<=0,99	1	>1	<=0,99	1	>1	<=0,99	1	>1	<=0,99	1	>1	<=0,99	1	>1
Atlas	88	**1%**	9%	2%	2%	**35%**	6%	**1%**	22%	14%	**1%**	3%	3%	6%	69%	25%
BioSQL	45	0%	**4%**	0%	7%	**40%**	18%	0%	**4%**	13%	0%	**4%**	9%	7%	53%	40%
Castor	91	**1%**	**59%**	22%	0%	4%	2%	2%	3%	4%	0%	0%	**1%**	3%	67%	30%
SlashCode	68	**1%**	**35%**	15%	**1%**	13%	18%	0%	**1%**	9%	0%	0%	6%	3%	50%	47%
Zabbix	56	**2%**	**27%**	11%	0%	20%	16%	0%	7%	13%	0%	**2%**	4%	2%	55%	43%

PROBABILITY FOR A TABLE OF A TOPOLOGICAL CATEGORY TO HAVE CERTAIN SIZE SCALE (PERCENTAGES OVER TOTAL #TABLES OF EACH TOPOLOGICAL CATEGORY)

TOPOLOGICAL CATEGORY

	ISOLATED				**SOURCE**				**LOOKUP**				**INTERNAL**				**Aggregate per Size Scale Category**			
	Total #Tables	<=0,99	1	>1	Total #Tables	<=0,99	1	>1	Total #Tables	<=0,99	1	>1	Total #Tables	<=0,99	1	>1	Total #Tables	<=0,99	1	>1
Atlas	11	**9%**	**73%**	18%	38	**5%**	**82%**	13%	32	**3%**	**59%**	38%	7	**14%**	**43%**	**43%**	88	6%	69%	25%
BioSQL	2	0%	**100%**	0%	29	**10%**	**62%**	28%	8	0%	**25%**	**75%**	6	0%	**33%**	**67%**	45	7%	53%	40%
Castor	75	**1%**	**72%**	27%	6	0%	**67%**	**33%**	9	**22%**	33%	**44%**	1	0%	0%	**100%**	91	3%	67%	30%
SlashCode	35	**3%**	**69%**	29%	22	**5%**	41%	**55%**	7	0%	**14%**	**86%**	4	0%	0%	**100%**	68	3%	50%	47%
Zabbix	22	**5%**	**68%**	27%	20	0%	**55%**	**45%**	11	0%	**36%**	**64%**	3	0%	**33%**	**67%**	56	2%	55%	43%

Figure 4.32 Top: Distribution of Tables over Topological and Size Scale Categories; Bottom: Probability for a Table to Have a Certain Size Scale

The lower part of Figure 4.32 contains the probability for a table of a certain topological category to experience a specific change in its size. Once again, the red and blue colors signify the most and the least populated groups of tables respectively, within each topological category without taking into account the total absence of including tables.

The similarities with regard to the combination of topological and size scale categories we identified among datasets can be summarized as follows:

- In all datasets, the absolute majority of the *isolated* tables remain *steady* in terms of their size.
- In four of the five datasets, the probability that a *source* table remains *steady* exceeds the value of 55%.
- Contrary to the behavior of the *source* tables, the *lookup* tables demonstrate a proclivity for increasing their attributes. This observation, which holds in four of the five datasets, except for Atlas, gives us an insight into the observation we briefly mentioned in the beginning of this subsection about the intense activity of the *lookup* tables. We can claim that, at least in four datasets, *lookup* tables' heightened update activity results in the expansion of their size.
- As regards the *internal* tables, in all the datasets it is highly likely that they undergo an expansion of their size during their existence in their datasets.
- Compared to the average probability of experiencing a certain size change, we distinguished two different patterns: the one according to which the *isolated* and *source* tables follow the average probability for size reduction, have higher probability for size steadiness and lower for size expansion and the other including *lookup* and *internal* tables with a potential for size reduction lower than the average with few exceptions, a probability for size steadiness below the average and a higher likelihood for size expansion.

Despite the patterns we observed with reference to the size scale of tables within each topological category, the evidence derived from the statistical tests are inadequate to support the correlation between the topological categories and the size scale ones. We utilized a 4x3 contingency table with its rows consisting of the topological categories and its columns representing the size scale categories. Apart from Castor and SlashCode for which the tests returned p-values 1.41E-02 and 6.8E-03 respectively, the statistical results for the rest of the datasets surpass the limit of 5% with the lowest p-value to be 0.089 in the case of the Atlas dataset.

In a nutshell, we distinguished two different behaviors concerning tables' size change and topological categories. The majority of the isolated and source tables remain steady, whereas the lookup and internal tables tend to increase their size.

4.5 Summary of Findings

In this chapter, our main objective was to study to what extent tables' topology can determine their evolutionary activity. Given that, we defined four topological categories at first based on the topology of the Diachronic Graph. We then used the schema histories of five open-source datasets to classify their tables into the topological categories and examined whether these categories are related with various measures of tables' activity. The labeling process posed the dilemma of how to handle tables that change topological categories throughout their lives and for this reason we manually inspected the changes of these tables. This manual examination led to a set of rules that applied to the tables' history would remove parts that would be confusing for the understanding of the true nature of the tables and would make feasible the automation of the classification process. However, we opted for utilizing the labels derived from the manual classification of the tables. Having assigned a single label to each table, we studied how the topological category of a table is related to various measures of its evolutionary activity, including duration, survival potential, birth version, update activity and the scale of its size change. The remainder of this section includes the most important findings concerning our study on tables' topology and evolution.

Concerning the normalized duration of the tables, we noticed that tables of long duration constitute the most popular group in four of the five datasets and those of short duration are the second largest category without exception among datasets. Studying the relationship between the topological category of a table and its normalized duration, we observed that if we ignore the existence of the *isolated* tables, the distributions of the *source* and *lookup* tables over the duration categories follow the average distributions, with the exceptions of BioSQL and Zabbix. We also identify the following interesting similarities among the datasets:

- In all datasets, *lookup* tables are prone to lives of long duration.

- In four of the five datasets, at least half of the *source* tables are long lived, apart from those of the BioSQL dataset.
- The *internal* tables avoid lives of short or medium duration, with the exception of BioSQL.
- As for the *isolated* tables, they avoid living for long periods, except for those of the Castor dataset.

The inclination of the tables towards lives of long duration holds for three of the four topological categories with few exceptions and this is an indication that it is quite unlikely that the topological categories are associated with tables' duration. This was also confirmed by the statistical tests we conducted for assessing the independence of tables' duration from their topological categories.

As far as survivors' distribution over the topological categories is concerned, we identified a *monotone decrease* pattern in the size of the categories' populations, starting from the *source* tables followed by *lookup* and ending with the *internal* tables in all datasets, except Castor. As for the relationship between the topological categories and the survival, we observed that the corresponding percentages are high in all datasets, excluding the *isolated* tables of the BioSQL dataset. The only difference between the topological categories is that the survival rate for the *source* and *lookup* tables follows the aggregate one, while in case of the *internal* tables the respective percentages are higher compared to the aggregate ones. The statistical evidence produced by the Chi-square and Fisher tests was not adequate to verify that topological categories can determine the survival rate of the including tables.

As regards the relationship between topological categories and tables' "birth" version, we were specifically interested to examine if the topological category of a table can have an effect on the probability to be introduced in the originating version of its dataset's history. Concerning the overall percentages of the tables "born" in the very first version of their datasets and excluding tables with no edges, we set apart the high portions of tables in three datasets, namely Atlas, Castor and SlashCode, in which the relative percentages exceed the value of 70%. The findings with reference to the relationship between the topological categories and the "birth" version can be summarized as follows:

- The *internal* and *lookup* tables demonstrate high probability to exist in the first version of their datasets, with the involved percentages of the former reaching the value of 100% in three of the five datasets.

- The *source* tables in all datasets, except BioSQL, present the second lowest potential for being born in the originating version of their datasets after the *isolated* tables.

- Compared to the aggregate probability of being created in the first version of a dataset and ignoring *isolated* category, the *source* and *lookup* tables approach the overall potential, while *internal* significantly exceed it.

We attributed the high probability for a *lookup* or *internal* table to be "born" in the first version to the *gravitation to rigidity* pattern, according to which it is not preferable to creating tables that attract foreign key constraints in later versions of the schema history. The statistical tests we performed were to some extent in favor of the relationship between one table's topological category and the probability of being created in the originating version.

Concerning tables' update activity and its relationship with topological categories, we initially classified tables with respect to their update profile in three categories, which are the *rigid* with no changes, the *quiet* with few updates and the *active* with more than five updates. We saw that the majority of the tables in four datasets are those with few changes, though we observed an increase in the number of the *rigid* tables after including tables of the *isolated* category. As for the distribution of the tables with reference to the topological categories and their update profile, we highlighted the following observations:

- Concerning the largest groups of tables, we encountered an inconsistent behavior, with *source* tables with a *quiet* update profile being the most popular in Atlas and Zabbix, with the *isolated* tables with no or few updates constituting the most multitudinous categories in Castor and SlashCode and with the *source* tables with all kinds of updates being the most populated groups in BioSQL.

- The least popular groups of tables are those of the *internal* category with no or few updates.

After that, we quantified the probability for a table of a certain topological category to develop a certain update activity. The most noteworthy findings are presented in the upcoming list:

- *Isolated* tables experience no or few updates during their lives.

- Apart from BioSQL, the *source* tables are very likely to sustain no or few updates.

- In three of the five datasets, *lookup* tables are subjects to few or many updates.
- In four of the five datasets, *internal* tables are expected to undergo many updates throughout their lives.

Concerning the potential for a table of certain update profile to belong to a specific topological category, we noticed that *rigid* tables are possible to belong to the categories of the *isolated* and *source* tables with a probability that is greater than 76%. *Quiet* tables are likely to be *source* in three of the five datasets, with the exceptions of Castor and SlashCode, in which *quiet* tables tend to be *isolated*. As regards *active* tables, there is not a consistent tendency among the datasets, except for the datasets of Castor and SlashCode, where the odds for *active* tables to be *isolated*, *source* or *lookup* are even.

The statistical tests we conducted for the relationship between topological categories and update activity returned low p-values for four of the five datasets and that makes us believe that there is a correlation between tables' topology and their update profile.

In the last part of our study we examined whether topological categories are related with the changes in the size of the tables. We group tables in three categories with respect to the change of their sizes between their first and last versions. We use the term *scale down* for tables that undergo a size reduction, the term *steady* for those with no change in their size and the term *scale up* for tables with a size expansion. We saw, on the one hand, the absolute majority of tables remain steady in terms of their size and a large portion increase their size and, on the other hand, tables that experience a size reduction to account for no more than 10% of the total number of tables in each dataset. Taking into consideration the topological categories of the tables, we end up with the following commonalities among the datasets:

- In three of the five datasets, the *isolated* tables with *steady* size create the largest groups with respect to the entire population of the tables. The only exceptions to that pattern are Atlas and BioSQL, in which *source* tables with *steady* size are the most populated group of tables.
- As for the least popular groups of tables, *internal* tables with all kinds of size changes along with tables of the other categories with size reduction or steadiness form groups whose cardinality does not surpass the 4% of the total number of tables per dataset.

Concerning the probability for a table of a specific topological category to go through a certain size change, we briefly describe the similarities we encountered as follows:

- The absolute majority of the *isolated* tables remain *steady*.
- In four of the five datasets, the probability for a *source* table to remain *steady* is greater than 55%.
- In four of the five datasets, *lookup* tables are prone to size expansion, which is not at all what one would expect since their size expansion is likely to affect tables that depend upon them.
- For the *internal* tables, it is likely (specifically, the least probability is 43%) that they will increase their size and it is highly improbable that they will end up with less attributes than those they consisted of in their first known version.

The results returned from the statistical tests we implemented are not adequate to reject the null hypothesis on the independence of tables' size changes from their topological categories. Nevertheless, we distinguished two different behaviors, the one of the *isolated* and *source* tables associated with a tendency towards not changing their size and the second one concerning *lookup* and *internal* tables that represents an inclination for size expansion.

Altogether, having conducted an in-depth survey concerning the impact of tables' topological categories on various measures of their evolutionary activity, we ended up with various findings, with the most significant being the correlations of topological categories with "birth" version as well as with update activity. These relationships were also confirmed by the statistical tests we conducted in order to evaluate to what extent the metrics of tables' evolution are related to their topological categories. As for the rest of the measures and their relationships with the topological categories, although we highlighted a few patterns among the datasets, the statistical evidence was not sufficient in order to support the existence of a statistically significant correlation between tables' topology and the measures of their evolution.

CHAPTER 5.

EXPORTING PARMENIDIAN TRUTH AS A WEB APPLICATION

The refactoring process of the Parmenidian Truth tool aimed at creating a project that will be incorporated easily in any other project providing all its functionalities through an interface. In this chapter, we exploit the new design of Parmenidian Truth tool to create a web application that will make possible for a user to visualize the evolution of a database's schema by running the application on a server. The first section of this chapter gives the necessary background on the architecture of a web application describing its main components and their roles. The second section presents the design of the web application that utilizes the functionalities of the Parmenidian Truth tool to analyze the schema evolution of a database.

5.1 Architecture of a Web Application

A *web application* enables the execution of an application resided in a server via the web. The users of the application exploit the client/server model sending requests to the server and receiving responses from the server. Figure

5.1 depicts the client/server model which we use to create the Parmenidian Truth web application.

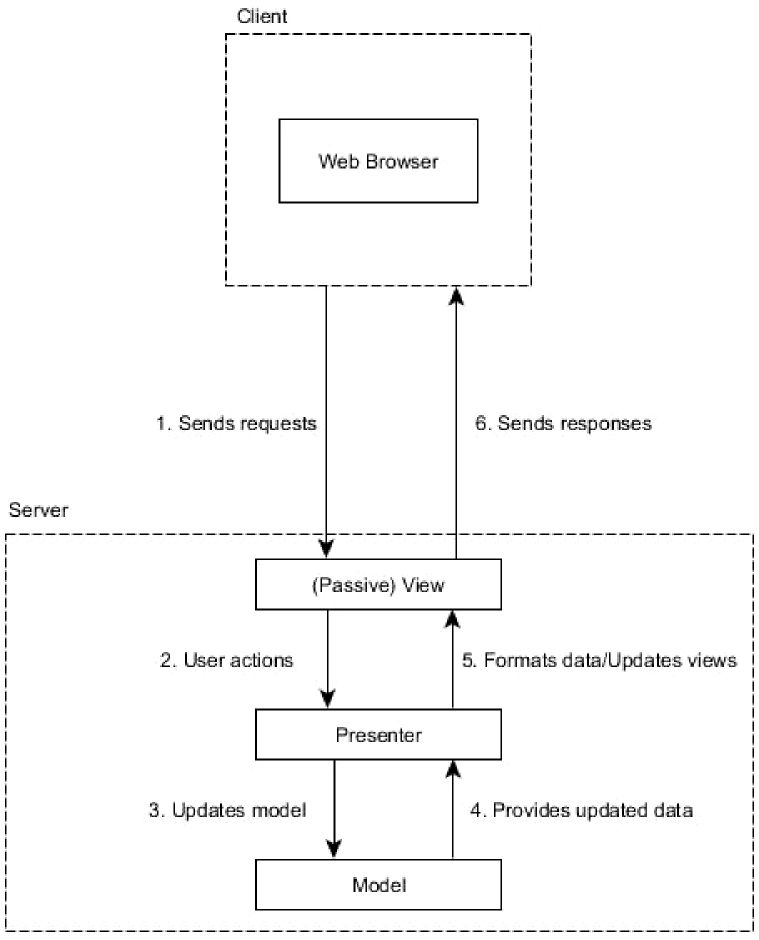


Figure 5.1 Client/Server Communication Model

The architecture of the server application is based on the *Model-View-Presenter* (MVP) design pattern. This pattern, which is derivative of the well-known *Model-View-Controller* (MVC) design model [KrPo88], was first introduced in [Pote96] in which author proposes a three-part decomposition of an application into the following components:

- *Model*: represents the domain of the application that includes the main data structures.
- *View*: includes every Graphical User Interface (GUI) utilized to present data provided by the presenter.

- *Presenter*: retrieves data from *model* component and formats it to be displayed in the *view* component.

The essential difference between the MVP and the MVC patterns is that in the former the *view* component has the passive role of displaying data and delegating user requests to the *presenter* component, while in the latter the *view* component updates itself whenever the *model* part changes. In other words, in MVP model there is no direct dependency between the *view* and the *model* components with their communication being implemented via the *presenter* part.

In our application, the role of the *model* component is assigned to java classes that make use of the Parmenidian Truth tool's functionalities and create objects that facilitate the presentation of the data in the *view* part of the application. The *view* component consists of *JavaServer Pages* (JSP) that allow users to develop web pages with dynamic content along with the static one, like that of HTML markup language. In this component, we also exploit the JavaScript language to incorporate the D3 library [BoHO11], a JavaScript library for data visualization. As for the *presenter* part, it consists of java *servlets* and classes, which receive clients' requests, communicate with the *model* module and update the *view* elements.

5.2 Design of Parmenidian Truth Web Application

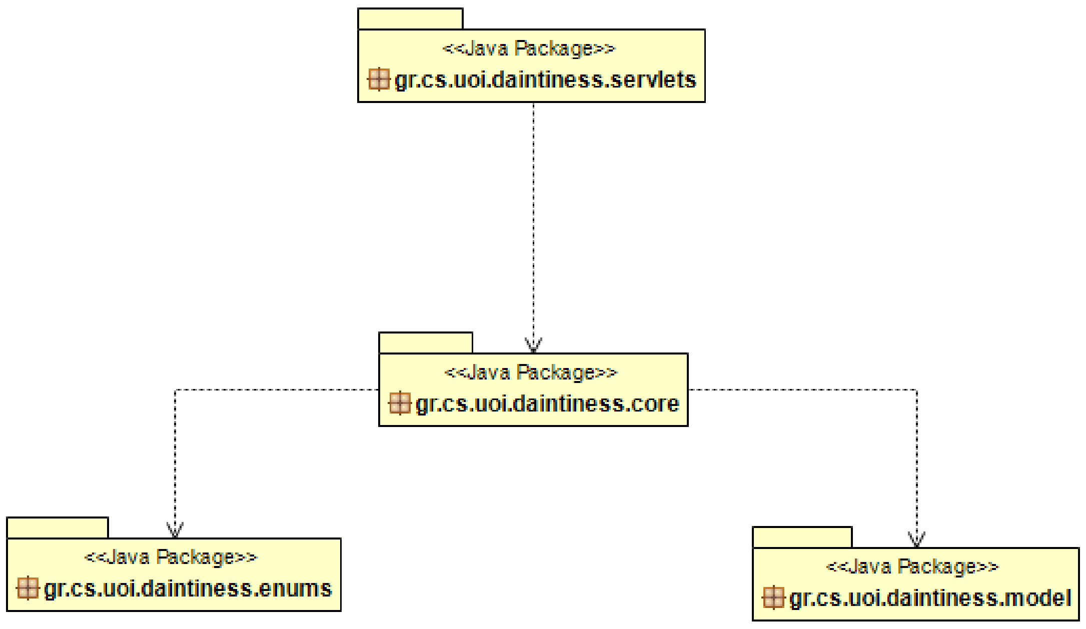
Apart from the existing functionalities provided by the Parmenidian Truth tool, we enriched our application with new ones that visualize the patterns presented in [VaZS15], in order to acquire a better view of whether and how evolution-related metrics are related to tables' properties. We also accompany the visualization results with a set of statistics that give an overview of the relationships between the measures and the properties previously mentioned. In a nutshell, the main functionalities provided by our web application are summarized as follows:

- Create/load a project in/from the server.
- Visualize Diachronic Graph/versions/evolution-related patterns.
- Compute evolution-related statistics.

5.2.1 Package Diagram

Figure 5.2 shows the package diagram of the web application's java resources, consisted of the following packages:

- *Servlets*: classes that are responsible for receiving clients' requests and sending responses back to them, playing along with the modules of the *core* package the role of an MVP *presenter*.
- *Core*: classes and interfaces that define the main functionalities of the application.
- *Model*: classes that represent the data structures of the application.
- *Enums*: enum types that help us to implement sets of predefined constants representing different categories of tables.



<<Java Package>>
gr.cs.uoi.daintiness.servlets
<<Java Package>>
gr.cs.uoi.daintiness.core
<<Java Package>>
gr.cs.uoi.daintiness.enums
<<Java Package>>
gr.cs.uoi.daintiness.model

Figure 5.2 Package Diagram of the Application's Java Resources

5.2.2 Class Diagrams

In this subsection, we present the including classes of the aforementioned packages of the web application. We do not include the class diagrams of the Parmenidian Truth tool, since we consider it as an independent project whose functionalities, presented in Chapter 3, we utilize to create the web application.

A. Servlets package

This package includes three classes each of which is responsible for receiving clients' requests related to the main functionalities provided by the application. Figure 5.3 depicts the class diagram of the *servlets* package along with the interfaces of the *core* package that they use.

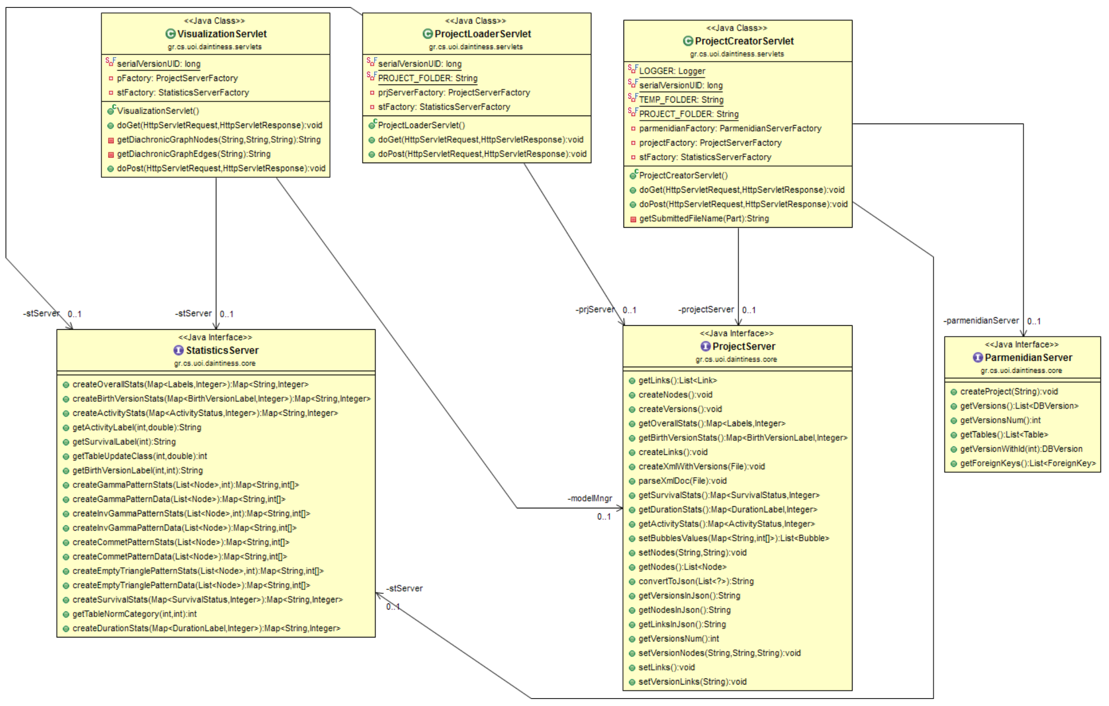

<<Java Class>>
VisualizationServlet
gr.cs.uoi.daintiness.servlets
serialVersionUID: long
pFactory: ProjectServerFactory
stFactory: StatisticsServerFactory
VisualizationServlet()
doGet(HttpServletRequest,HttpServletResponse):void
getDiachronicGraphNodes(String,String,String):String
getDiachronicGraphEdges(String):String
doPost(HttpServletRequest,HttpServletResponse):void
<<Java Class>>
ProjectLoaderServlet
gr.cs.uoi.daintiness.servlets
serialVersionUID: long
PROJECT_FOLDER: String
prjServerFactory: ProjectServerFactory
stFactory: StatisticsServerFactory
ProjectLoaderServlet()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void
<<Java Class>>
ProjectCreatorServlet
gr.cs.uoi.daintiness.servlets
LOGGER: Logger
serialVersionUID: long
TEMP_FOLDER: String
PROJECT_FOLDER: String
parmenidianFactory: ParmenidianServerFactory
projectFactory: ProjectServerFactory
stFactory: StatisticsServerFactory
ProjectCreatorServlet()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void
getSubmittedFileName(Part):String
-stServer 0..1
-stServer 0..1
-prjServer 0..1
-projectServer 0..1
-parmenidianServer 0..1
-modelMngr 0..1
-stServer 0..1
<<Java Interface>>
StatisticsServer
gr.cs.uoi.daintiness.core
createOverallStats(Map<Labels,Integer>):Map<String,Integer>
createBirthVersionStats(Map<BirthVersionLabel,Integer>):Map<String,Integer>
createActivityStats(Map<ActivityStatus,Integer>):Map<String,Integer>
getActivityLabel(int,double):String
getSurvivalLabel(int):String
getTableUpdateClass(int,double):int
getBirthVersionLabel(int,int):String
createGammaPatternStats(List<Node>,int):Map<String,int[]>
createGammaPatternData(List<Node>):Map<String,int[]>
createInvGammaPatternStats(List<Node>,int):Map<String,int[]>
createInvGammaPatternData(List<Node>):Map<String,int[]>
createCommetPatternStats(List<Node>):Map<String,int[]>
createCommetPatternData(List<Node>):Map<String,int[]>
createEmptyTrianglePatternStats(List<Node>,int):Map<String,int[]>
createEmptyTrianglePatternData(List<Node>):Map<String,int[]>
createSurvivalStats(Map<SurvivalStatus,Integer>):Map<String,Integer>
getTableNormCategory(int,int):int
createDurationStats(Map<DurationLabel,Integer>):Map<String,Integer>
<<Java Interface>>
ProjectServer
gr.cs.uoi.daintiness.core
getLinks():List<Link>
createNodes():void
createVersions():void
getOverallStats():Map<Labels,Integer>
getBirthVersionStats():Map<BirthVersionLabel,Integer>
createLinks():void
createXmlWithVersions(File):void
parseXmlDoc(File):void
getSurvivalStats():Map<SurvivalStatus,Integer>
getDurationStats():Map<DurationLabel,Integer>
getActivityStats():Map<ActivityStatus,Integer>
setBubblesValues(Map<String,int[]>):List<Bubble>
setNodes(String,String):void
getNodes():List<Node>
convertToJson(List<?>):String
getVersionsInJson():String
getNodesInJson():String
getLinksInJson():String
getVersionsNum():int
setVersionNodes(String,String,String):void
setLinks():void
setVersionLinks(String):void
<<Java Interface>>
ParmenidianServer
gr.cs.uoi.daintiness.core
createProject(String):void
getVersions():List<DBVersion>
getVersionsNum():int
getTables():List<Table>
getVersionWithId(int):DBVersion
getForeignKeys():List<ForeignKey>

Figure 5.3 Class Diagram of the Servlets Package (along with the Interfaces of the Core Package)

The absence of relationships between the classes of the *servlets* package is due to the fact that each of them serves different types of requests. More specifically, the *ProjectCreatorServlet* class is responsible for receiving requests for the creation of a new project. It then receives a set of data definition files containing database's history and by utilizing Parmenidian Truth tool as well as the classes of the core package a model of the schema evolution is created. It finally sends back to application's front-end information related to the evolution of the database's schema. The *ProjectLoaderServlet* class utilizes the aforementioned model that contains the required information for implementing all the functionalities our application provides without the necessity of submitting successive requests to the Parmenidian Truth tool. The

VisualizationServlet class processes requests concerning the visualization of the patterns, presented in [VaZS15], the Diachronic Graph and the versions it consists of. As mentioned in Chapter 3, the *Diachronic Graph* is a graph with its nodes and edges corresponding to database's tables and foreign keys, respectively.

B. *Core package*

The *core* package implements the business logic part of the application via retrieving the data provided by the *model* package, processing them and producing the responses to the clients' requests. It consists of three interfaces, namely *ProjectServer*, *ParmenidianServer* and *StatisticsServer* which offer the required methods for implementing the main functionalities of the application.

The *ProjectServer* interface has the central role of serving the classes of the *servlets* package via providing all the necessary data derived from the *model* package. The *ParmenidianServer* interface defines the methods that offer the data derived from the Parmenidian Truth tool. The *StatisticsServer* interface comprises the methods that process the data and create a set of statistical information that summarize the evolution of the database's schema. Apart from the interfaces, there also exist two classes, the *JsonConverter* and *Bubble* classes, which convert data in a format that will facilitate their visualization. Figure 5.4 presents the participating classes and the dependencies between them in the *core* package.

C. *Model package*

This package encompasses the domain classes of the application that model the versions, the tables and the foreign keys of a database. It also includes the *Model* module that represents the data needed to load a project and the *XmlProvider* class used to convert the objects of the *Model* class to data in xml format and vice versa. Figure 5.5 illustrates the components of the *model* package.

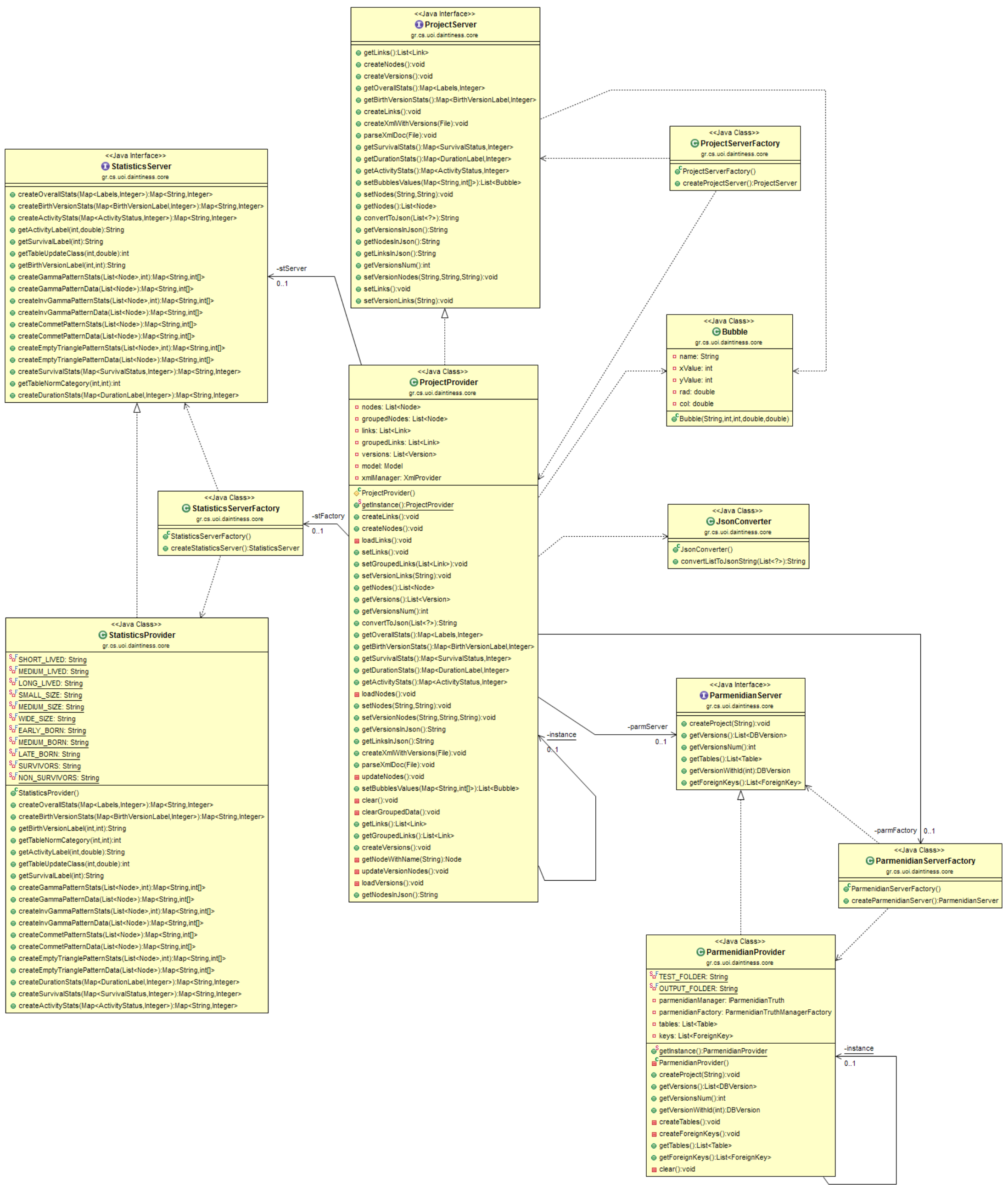

<<Java Interface>>
ProjectServer
gr.cs.uoi.daintiness.core
getLinks():List<Link>
createNodes():void
createVersions():void
getOverallStats():Map<Labels,Integer>
getBirthVersionStats():Map<BirthVersionLabel,Integer>
createLinks():void
createXmlWithVersions(File):void
parseXmlDoc(File):void
getSurvivalStats():Map<SurvivalStatus,Integer>
getDurationStats():Map<DurationLabel,Integer>
getActivityStats():Map<ActivityStatus,Integer>
setBubblesValues(Map<String,int[]>):List<Bubble>
setNodes(String,String):void
getNodes():List<Node>
convertToJson(List<?>):String
getVersionsInJson():String
getNodesInJson():String
getLinksInJson():String
getVersionsNum():int
setVersionNodes(String,String,String):void
setLinks():void
setVersionLinks(String):void
<<Java Class>>
ProjectServerFactory
gr.cs.uoi.daintiness.core
ProjectServerFactory()
createProjectServer():ProjectServer
<<Java Interface>>
StatisticsServer
gr.cs.uoi.daintiness.core
createOverallStats(Map<Labels,Integer>):Map<String,Integer>
createBirthVersionStats(Map<BirthVersionLabel,Integer>):Map<String,Integer>
createActivityStats(Map<ActivityStatus,Integer>):Map<String,Integer>
getActivityLabel(int,double):String
getSurvivalLabel(int):String
getTableUpdateClass(int,double):int
getBirthVersionLabel(int,int):String
createGammaPatternStats(List<Node>,int):Map<String,int[]>
createGammaPatternData(List<Node>):Map<String,int[]>
createInvGammaPatternStats(List<Node>,int):Map<String,int[]>
createInvGammaPatternData(List<Node>):Map<String,int[]>
createCommetPatternStats(List<Node>):Map<String,int[]>
createCommetPatternData(List<Node>):Map<String,int[]>
createEmptyTrianglePatternStats(List<Node>,int):Map<String,int[]>
createEmptyTrianglePatternData(List<Node>):Map<String,int[]>
createSurvivalStats(Map<SurvivalStatus,Integer>):Map<String,Integer>
getTableNormCategory(int,int):int
createDurationStats(Map<DurationLabel,Integer>):Map<String,Integer>
-stServer
0..1
<<Java Class>>
Bubble
gr.cs.uoi.daintiness.core
name: String
xValue: int
yValue: int
rad: double
col: double
Bubble(String,int,int,double,double)
<<Java Class>>
ProjectProvider
gr.cs.uoi.daintiness.core
nodes: List<Node>
groupedNodes: List<Node>
links: List<Link>
groupedLinks: List<Link>
versions: List<Version>
model: Model
xmlManager: XmlProvider
ProjectProvider()
getInstance():ProjectProvider
createLinks():void
createNodes():void
loadLinks():void
setLinks():void
setGroupedLinks(List<Link>):void
setVersionLinks(String):void
getNodes():List<Node>
getVersions():List<Version>
getVersionsNum():int
convertToJson(List<?>):String
getOverallStats():Map<Labels,Integer>
getBirthVersionStats():Map<BirthVersionLabel,Integer>
getSurvivalStats():Map<SurvivalStatus,Integer>
getDurationStats():Map<DurationLabel,Integer>
getActivityStats():Map<ActivityStatus,Integer>
loadNodes():void
setNodes(String,String):void
setVersionNodes(String,String,String):void
getVersionsInJson():String
getLinksInJson():String
createXmlWithVersions(File):void
parseXmlDoc(File):void
updateNodes():void
setBubblesValues(Map<String,int[]>):List<Bubble>
clear():void
clearGroupedData():void
getLinks():List<Link>
getGroupedLinks():List<Link>
createVersions():void
getNodeWithName(String):Node
updateVersionNodes():void
loadVersions():void
getNodesInJson():String
<<Java Class>>
StatisticsServerFactory
gr.cs.uoi.daintiness.core
StatisticsServerFactory()
createStatisticsServer():StatisticsServer
-stFactory
0..1
<<Java Class>>
JsonConverter
gr.cs.uoi.daintiness.core
JsonConverter()
convertListToJsonString(List<?>):String
<<Java Class>>
StatisticsProvider
gr.cs.uoi.daintiness.core
SHORT_LIVED: String
MEDIUM_LIVED: String
LONG_LIVED: String
SMALL_SIZE: String
MEDIUM_SIZE: String
WIDE_SIZE: String
EARLY_BORN: String
MEDIUM_BORN: String
LATE_BORN: String
SURVIVORS: String
NON_SURVIVORS: String
StatisticsProvider()
createOverallStats(Map<Labels,Integer>):Map<String,Integer>
createBirthVersionStats(Map<BirthVersionLabel,Integer>):Map<String,Integer>
getBirthVersionLabel(int,int):String
getTableNormCategory(int,int):int
getActivityLabel(int,double):String
getTableUpdateClass(int,double):int
getSurvivalLabel(int):String
createGammaPatternStats(List<Node>,int):Map<String,int[]>
createGammaPatternData(List<Node>):Map<String,int[]>
createInvGammaPatternStats(List<Node>,int):Map<String,int[]>
createInvGammaPatternData(List<Node>):Map<String,int[]>
createCommetPatternStats(List<Node>):Map<String,int[]>
createCommetPatternData(List<Node>):Map<String,int[]>
createEmptyTrianglePatternStats(List<Node>,int):Map<String,int[]>
createEmptyTrianglePatternData(List<Node>):Map<String,int[]>
createDurationStats(Map<DurationLabel,Integer>):Map<String,Integer>
createSurvivalStats(Map<SurvivalStatus,Integer>):Map<String,Integer>
createActivityStats(Map<ActivityStatus,Integer>):Map<String,Integer>
-instance
0..1
-parmServer
0..1
<<Java Interface>>
ParmenidianServer
gr.cs.uoi.daintiness.core
createProject(String):void
getVersions():List<DBVersion>
getVersionsNum():int
getTables():List<Table>
getVersionWithId(int):DBVersion
getForeignKeys():List<ForeignKey>
-parmFactory
0..1
<<Java Class>>
ParmenidianServerFactory
gr.cs.uoi.daintiness.core
ParmenidianServerFactory()
createParmenidianServer():ParmenidianServer
<<Java Class>>
ParmenidianProvider
gr.cs.uoi.daintiness.core
TEST_FOLDER: String
OUTPUT_FOLDER: String
parmenidianManager: IParmenidianTruth
parmenidianFactory: ParmenidianTruthManagerFactory
tables: List<Table>
keys: List<ForeignKey>
getInstance():ParmenidianProvider
ParmenidianProvider()
createProject(String):void
getVersions():List<DBVersion>
getVersionsNum():int
getVersionWithId(int):DBVersion
createTables():void
createForeignKeys():void
getTables():List<Table>
getForeignKeys():List<ForeignKey>
clear():void
-instance
0..1

Figure 5.4 Class Diagram of the Core Package

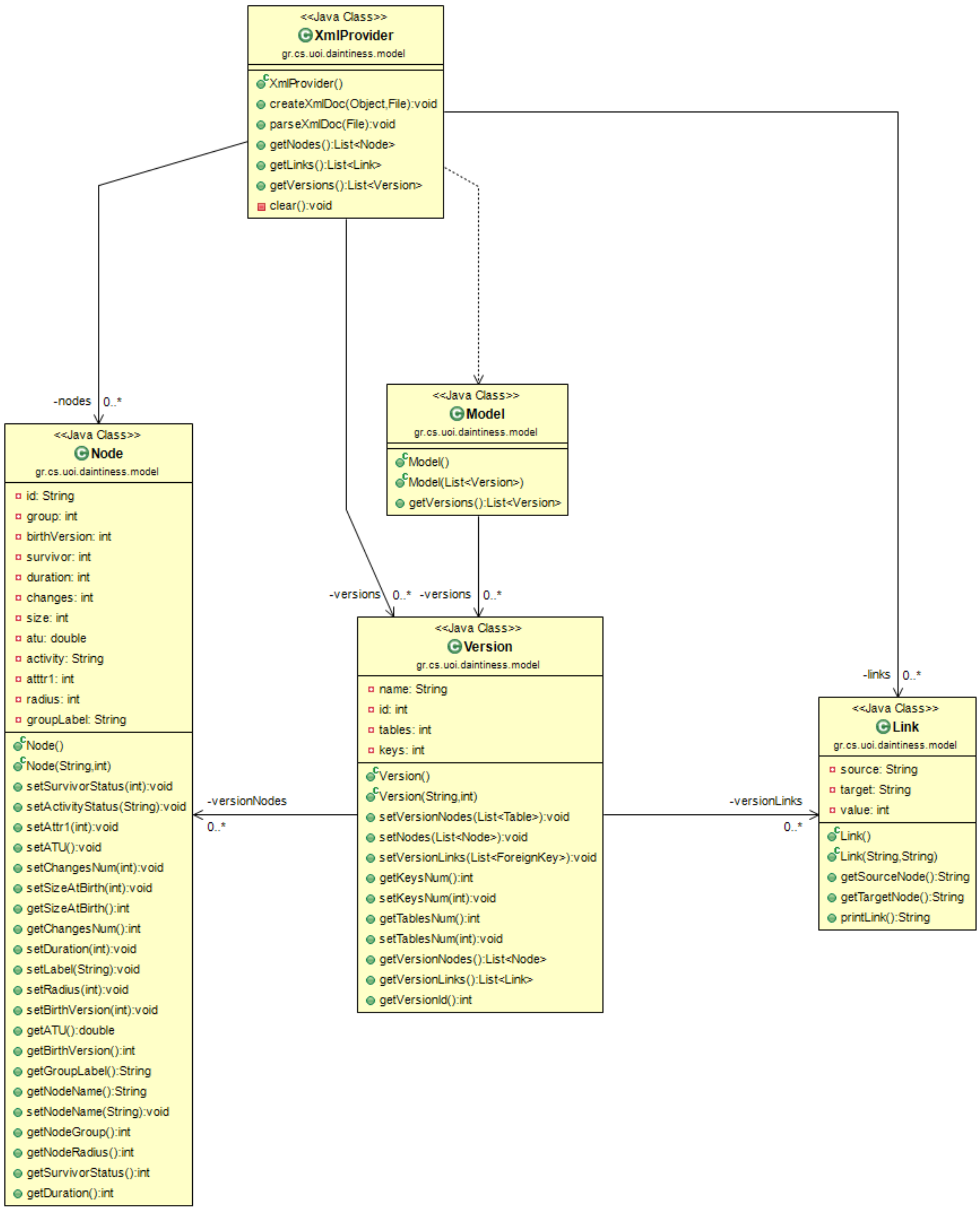


Figure 5.5 Class Diagram of the Model Package

D. *Enums package*

The enum types included in this package define tables' categories based on their activity profile, their originating version, their duration in schema history and their survival status based on their existence in the last version of database's schema. These types are used to quantify tables' distribution over the different categories and to facilitate the visualization of the Diachronic Graph. Figure 5.6 depicts the class diagram of the enums package.

The *ActivityStatus* type defines three table categories based on tables' update activity during their existence in the database. The *DurationLabel* type classifies tables in terms of their normalized duration, which is the number of the versions they exist over the total number of the versions. The *BirthVersionLabel* type includes three constants corresponding to the different tables' categories pertaining to their birth versions. The *SurvivalStatus* type assigns to each table a label based on whether or not it exists in the last version of database's schema history. The last type, *Labels*, is only used to provide the including constants as labels to the methods that compute the evolution-related statistical information.

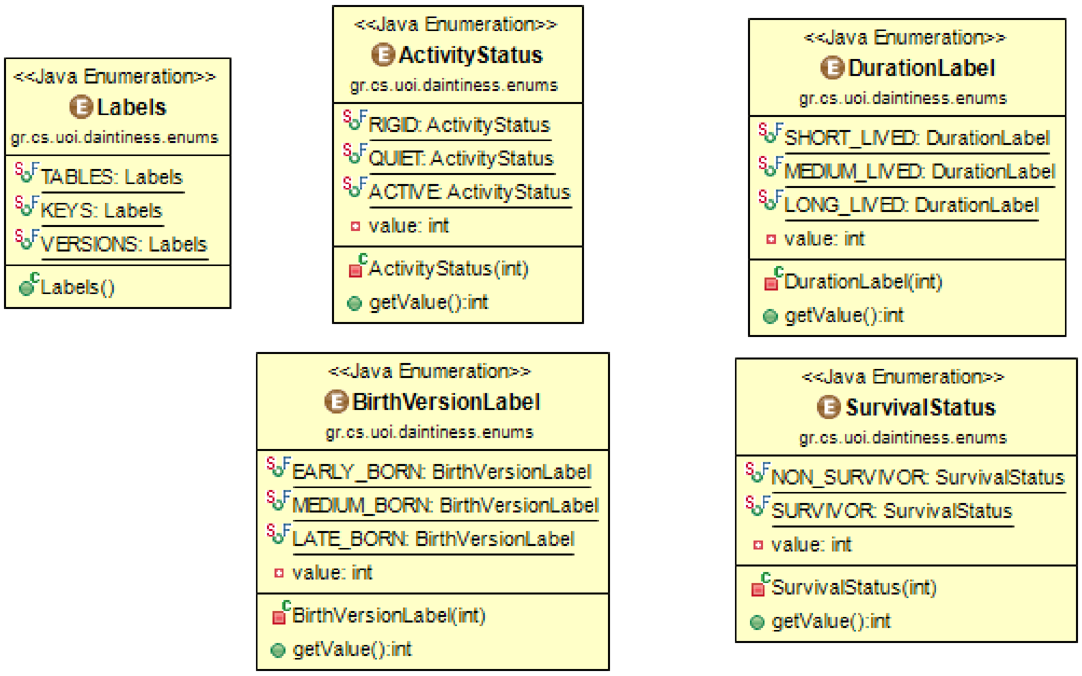


Figure 5.6 Class Diagram of the Enums Package

CHAPTER 6.

CONCLUSIONS AND FUTURE WORK

The final chapter of the current thesis summarizes the major findings of our study, outlines the answers to the research questions we stated in the introductory chapter and finally suggests potential issues for future work.

6.1 Conclusions

The twofold objective of this thesis was: (a) to examine whether there is a correlation between tables' topological properties and various metrics of their evolutionary activity and (b) to improve the internal quality of an existing tool for the study of schema evolution with respect to foreign keys, by introducing principled rectification mechanisms and applying a set of recommended refactoring patterns. Thus, we conducted an in-depth analysis by classifying tables in four topological categories, namely isolated, source, lookup and internal and then we study how these categories are likely to determine tables' durations, their potential to exist in the last version of the schema history, the probability for a table to exist in the very first version of their database, tables' update profile and their size change between the first and the last versions.

The most important findings that are also supported by the corresponding statistical tests were that there exist: (a) a relationship between tables' topological categories and the probability to be born in the originating version as well as (b) a correlation between tables' topology and their update activity. As for the former relationship, we identified two different behaviors among the topological categories, with the *lookup* and the *internal* tables demonstrating a proclivity for existing in the early, if not in the very first, versions of their database's history, while the *isolated* and the *source* tables are more likely to be introduced in versions succeeding the originating one. Concerning the latter correlation, we recognize a *monotone increase* pattern in the intense of tables' update activity with their topological complexity. Specifically, the *isolated* and the *source* tables are associated with no or few updates, the *lookup* tables with few or many updates and the *internal* tables with many changes. For the rest of the measures and their relationships with tables' topological categories although we pointed out several commonalities among the datasets examined, we could not present solid evidence that would verify the existence of these correlations.

We also presented a principled refactoring process applied in the Parmenidian Truth tool, which visualizes the schema evolution of relational databases. First, we inspected design defects that would complicate any expendability efforts and the reusability of the tool. For each defect, we applied the necessary modifications in tool's source code aiming at eliminating it and complying with the recommended design principles and patterns. For the modules we either modified or added, we created unit tests to verify that tool's expected behavior has not altered after the refactoring process. Finally, having performed all the refactoring actions, we assessed the improvements in the tool's architecture derived from the restructuring process.

Following the refactoring process, we utilized the modified Parmenidian Truth tool to create a web application that comprises an alternative solution for visualizing the evolution of databases' schemata and quantifying the respective statistical information.

6.2 Future work

To the best of our knowledge, this was the first work that revolved around the relationship of tables' involvement with foreign keys with their evolutionary behavior. In a follow-up work, one can investigate if a different topological classification of the tables would lead to different conclusions on the role of tables' topology in the schema evolution. In our study, we chose to assign a single label to each table, since the single label scheme facilitated our goal to associate the topological labels with the evolutionary measures. A multi-labeling scheme that does not ignore the label changes that a table experiences is likely to reveal evolutionary features that our work neglected.

The second issue that can be the objective of future research has to do with what we call a "*second-pass*" in tables' classification process. In our work we assign a label to each table based on its inciting edges without considering the labels of its adjacent tables. It would be interesting to see, after classifying tables via the process we proposed, whether a second classification phase that takes into account tables' neighborhood, practically resulting in a different set of topological categories, would provide us with new information about the ways tables evolve with respect to the topological categories they belong to.

BIBLIOGRAPHY

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. Refactoring: Improving the Design of Existing Code, pp. 46-47, 63-72, Addison Wesley Longman, Inc, 1999.

[MaMa06] R. C. Martin, M. Martin. Agile Principles, Patterns, and Practices in C#, pp. 155, Prentice Hall, 2006.

[Mart00] R. C. Martin. Design Principles and Design Patterns, pp. 14-16, 24-26, Objectmentor.com, 2000.

[ChKe92] S. R. Chidamper, C. F. Kemerer. A Metrics Suite for Object Oriented Design, pp. 19-21, 24-27, Center for Information Systems Research, Sloan School of Management, Massachusetts Institute of Technology, 1992.

[BeCu89] K. Beck, W. Cunningham. A Laboratory for Teaching Object Oriented Thinking, Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89), pp. 1-6, New Orleans, Louisiana, USA, October 1989.

[Fabe07] S. Faber. Mockito Framework. Available at HTTP://SITE.MOCKITO.ORG/ , 2007.

[Buga07] Bugan IT Consulting UG. Structure Analysis for Java. Available at HTTP://STAN4J.COM/ , 2007.

[Sjøb93] Dag Sjøberg. Quantifying Schema Evolution. Information and Software Technology, 35(1), pp. 35-44, January 1993.

[Kara02] Amela Karahasanovic. Identifying Impacts of Database Schema Changes on Applications. Available at HTTPS://WWW.RESEARCHGATE.NET/ , 2002.

[CuMZ08] Carlo A. Curino, Hyun J. Moon, Carlo Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench. Proceedings of VLDB Endowment, 1(1), pp. 761-772, August 2008.

[CMTZ08] Carlo A. Curino, Hyun J. Moon, Letizia Tanca, Carlo Zaniolo. Schema Evolution in Wikipedia: toward a Web Information System Benchmark. In Proc. of 10th International Conference on Enterprise Information Systems (ICEIS 2008), pp. 323-332, Barcelona, Spain, June 2008.

[WuNe11] Shengfeng Wu, Iulian Neamtiu. Schema Evolution Analysis for Embedded Databases. In Proc. 27th International Conference on Data Engineering Workshops (ICDEW 2011), pp. 151-155, Hannover, Germany, April 2011.

[PVSV12] G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou. Metrics for the Prediction of Evolution Impact in ETL Ecosystems: A Case Study. Journal on Data Semantics, 1(2), pp. 75-97, August 2012.

[QiLS13] D. Qiu, B. Li, Z. Su. An Empirical Analysis of the Co – evolution of Schema and Code in Database Applications. In Proc. of 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), pp. 125-135, Saint Petersburg, Russia, August 2013.

[CGMM+15] A. Cleve, M. Gobert, L. Meurice, J. Maes, J. Weber. Understanding Database Schema Evolution: A Case Study. Science of Computer Programming, 97(1), pp. 113-121, January 2015.

[VaZS15] P. Vassiliadis, A. V. Zarras, I. Skoulis. How is Life for a Table in an Evolving Relational Schema? Birth, Death and Everything In Between. In: Johannesson P., Lee M., Liddle S., Opdahl A., Pastor López Ó. (eds.) Conceptual Modeling ER 2015. LNCS, 9381, pp. 453-466, December 2015.

[VaZS17] P. Vassiliadis, A. V. Zarras, I. Skoulis. Gravitating to Rigidity: Patterns of Schema Evolution – and its Absence – in the Lives of Tables. Information Systems, 63, pp.24-46, January 2017.

[VaZa17] P. Vassiliadis, A. V. Zarras. Survival in Schema Evolution: Putting the Lives of Survivor and Dead Tables in Counterpoint. 29th International Conference on Advanced Information Systems Engineering (CAiSE 2017), Essen, Germany, June 2017.

[VKZZ17] P. Vassiliadis, M. Kolozoff, M. Zerva, A. V. Zarras. Schema Evolution and Foreign Keys: Birth, Eviction, Change and Absence. In Proc. of 36th International Conference on Conceptual Modeling (ER 2017), pp. 106-119, Valencia, Spain, November 2017.

[VKZZ19] P. Vassiliadis, M. Kolozoff, M. Zerva, A. V. Zarras. Schema Evolution and Foreign Keys: a Study on Usage, Heartbeat of Change and Relationship of Foreign Keys to Table Activity. Computing (2019), pp. 1-26, January 2019.

[KrPo88] G. E. Krasner, S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming (JOOP), 1(3), pp. 26-49, Aug./Sep. 1988.

[Pote96] M. Potel. MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java. Taligent Inc, 1996.

[BoHO11] M. Bostock, J. Heer, V. Ogievetsky. D3.js. Available at HTTPS://GITHUB.COM/D3/D3, 2011.

SHORT CV

Konstantinos Dimolikas was born in Ioannina, Greece. In 2013, he received his Diploma in Electrical & Computer Engineering from the National Technical University of Athens. After fulfilling his military service and for 8 months he worked as a Programmer-Analyst at the software development department of the Independent Power Transmission Operator (IPTO). In 2016, he started his graduate studies at the Department of Computer Science & Engineering in the University of Ioannina while working as a Programmer in the public sector.