2013 IEEE 16th International Conference on Computational Science and Engineering

Relational Database Design Patterns

Ramzi A. Haraty and Georges Stephan
Department of Computer Science and Mathematics
Lebanese American University
Beirut, Lebanon 1102-2801
Email: rharaty@lau.edu.lb

***Abstract*-- Design Patterns (DPs) recently took over software development by storm. Object oriented development, workflows and distributed computing are a few of the disciplines where DPs made a significant difference in delivery speed and product quality. Unfortunately, databases had not yet attracted the attention of DP designers. This work benefits from the growing popularity of open source-data centric and semantic web applications, and uncovers 24 DPs related to databases. These DPs will give insight to any database architect on how common real world entities are being modeled.**

Keywords - development process, database design, design pattern

I. INTRODUCTION

All software development processes, and especially for data centric applications, start with the gathering and analysis of the user's requirements. Many techniques were proposed [1][2], but all approaches are time and resource consuming and, therefore expensive. Moreover, software developers will not initiate this process unless the project is officially started. The time wasted through the long and iterative process of the gathering of business requirements could be better used. The idea behind this work originated from recent proliferation of open source applications. As many of these projects rely on a persistence engine, the idea is to gather and study the database design of the largest possible number of open source applications. Furthermore, the increasing popularity of design patterns (a design pattern is a common solution to a common problem found by different people) in many disciplines, and especially in software engineering, was a motivation to try to come up with Database Design Patterns or (DDP).

Till this day, DDP are still not recognized by the developers' community in the same manner as object oriented DPs. This fact is reinforced by the scarcity of papers, books, presentation and articles covering the subject. And the few papers that tackle this issue do it mainly for pedagogical purposes; while, in comparison, DPs have deeply penetrated enterprises. There are many reasons why DDPs are absent from the developers table of tools. First, no two authors agree on the definition of DDPs. Second, the de-facto standard database language (SQL) does not offer the same vocabulary and grammar richness as other programming languages, such as JAVA, C++ or Smalltalk. This gives the false impression that SQL offers fewer design possibilities; and that therefore, few things could go wrong. Finally, most authors seem to have forgotten the most attribute of a design pattern - it should be discovered! And the only way to that, is to look at HOW developers are designing their databases, not how they SHOULD do it. The only way to do that is to gather as much database designs as possible. The first place to look for would be in open source repositories, were hundreds of thousands of applications exist, and were code is freely shared among projects and repositories. Pattern discovery may lead to the discovery of anti-patterns. These are patterns that should not be used, and will be treated and documented just like other patterns. The challenge here is how to determine that the design is bad. In this work, a DDP will be considered as bad if it opens a major security risk, or if it corrupts data. The analysis of open source databases for the largest open source applications confirmed the existence of DDPs. It seems that no matter what is being modeled, the number of common design characteristics are often much larger than the differences.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 presents the methodology of finding the DDPs and documenting them. Section 4 concludes the paper.

II. RELATED WORK

The work carried out by the Gang of Four, or GOF, [3] is the initial spark that popularized the use of design patterns in the word of computer science, especially for object oriented development. After looking at the source code of many developers around the world, and after seeing how many people who never met or never communicated, solved the same problem in a similar method, the idea of software design patterns emerged. Although a concept might look or could be demonstrated as to be efficient, it could not be categorized as a design pattern if it is not implemented in existing software.

In [4], the authors present an approach for interacting with a database. The authors detected three database design patterns:

- Pivoting: This purpose of the pattern is to allow the user to easily switch/transform rows to columns and vice-versa. The attributes become rows instead of being columns, and each attribute is linked to a table that defines attribute types. This pattern is often used by workflow systems, and is referred to as the workflow pattern.

978-0-7695-5096-1/13 $31.00 © 2013 IEEE
DOI 10.1109/CSE.2013.124

- Materialization: This pattern allows the implementation of a father-child relationship as commonly found in object-oriented languages. The pattern establishes a one to one relationship between tables, so that the father classes (the equivalent of an abstract class) are linked to tables implementing the attributes of the child classes.
- Generalization and Specialization: This pattern implements the "is-a" concept commonly found in object oriented databases. For example, a cedar "is a" tree, which means that the interface or abstract class called "tree" is implemented or extended by a cedar. To implement this feature, each row in a table has a "parent id" column referring to the key of the father record in the father table in a "classes" table.

Of all the reviewed documents, [5] is the closest to our work. The difference is that the patterns are not "discovered." They are sourced from best practices, and then located in some ERP systems such as SAP. Their work focus on mainly two DPs:

- Internationalization: Allows an application to provide an interface with support to multiple languages.
- Accessible Transaction Log: All database engines with ACID (atomicity, consistency, isolation, durability) support need to keep a log of all the executed transactions. The image of the data to be changed is kept in the log, and is retrieved and re-applied in an executed query that is rolled back.

The author of [6] proposes new approaches to a semi-formal definition of DDPs. But instead of graphically represent EERs, the author proposed the following textual annotation:

- $A(a0, a1, \ldots, ak)$ denotes an entity type A with identifier *a0* and additional attributes $a1, \ldots, ak$.
- $R(A(ma, Ma), B(mb, Mb))$ denotes a binary relationship type R that relates entities A and B; A (respectively B) participates in R with minimum cardinality *ma* and maximum cardinality *Ma* (this also applies to B with *mb* and *Mb*).
- $A(a0, \ldots, ak) \rightarrow \{B1, \ldots, Bl\}$ denotes that A is a specialization of $B1, \ldots, B1$ with additional attributes $a0, ..., ak$; we write $(a0, ..., ak) \rightarrow B$ when A has only one parent.
- $A(a0, a1, ..., ak) \Rightarrow \{B1, ..., Bl\}$ denotes that A is a weak entity type identified by $B1, ..., Bl$, with (optional) semi-identifier *a0* and with additional attributes $a1, ..., ak$; we write $A(a0, ..., ak) \Rightarrow B$ when A is identified by a single entity type B.
- $<G>\!\!\rightarrowtail\!\!(A(ma, Ma), B(mb, Mb))$ denotes an aggregation of A and B. An aggregation may appear in the previous notations wherever an entity type occurs.

The author believes that DDPs should not be domain specific, and should not be complex in order to be efficiently re-used and understood by a large audience. Within these lines, the authors proposed two DDPs:

- Multiple Roles: This pattern describes the modeling of a relationship where an entity has multiple relationship types with another.
- Hierarchy: This pattern describes the modeling of a hierarchy where data items within the same table are related in a father-child fashion.

The author believes that this semi-formal representation would help document and enrich DDPs as well as the "impedance-mismatch" between object-oriented programing languages and relational databases.

The authors of [7] analyzed the efficiency of design patterns in large companies namely, Hewitt Associates, AT&T, Motorola, BNR and Siemens. The paper includes a separate analysis for each.

All the work reviewed above, agree that design patterns have a positive impact on software development cycles and very often on product quality. DPs significantly improve the communication among developers. Unfortunately, DDPs are only covered for pedagogical purposes. A more practical approach would greatly benefit database designers in the same way object oriented DPs are helping developers, as reported in all the reviewed work.

III. METHODOLOGY

Google offers a search engine called "Google code search" [8]. This search engine crawls through all the open source repositories and searches within the source code of applications. In order to locate all the database designs, the search was focused on all files with an ".SQL" extension containing the words "CREATE TABLE". The ideas here is that all data-centric application are distributed with a ".SQL" text file containing the queries that create the whole database, including the tables, the indexes, the triggers and the stored procedures. Google code search returned about 200 files. Further search performed on well-known repositories such as sourceforge.net [9] and github.com [10] yielded the retrieval of 800 additional files. We then moved to analyze these scripts to try to find any DDPs.

Many of the databases for open source applications are modeling objects or entities. These are often the same across applications: books, people, systems, documents, files, buildings, addresses, etc. By looking at all these designs, it becomes possible to take a table from one system and drop it onto another one by just changing the table labels or giving them aliases. This is known as semantic database modeling [11]. In this modeling a dictionary of real world entities is built, and each entity has all of its attributes and how they relate to other entities already defined. The developers will have to "pick" an object from the dictionary and integrate it in their designs. The reason why we do not see semantic database modeling in any of the open source applications is that the proposed design is too complex and cannot apply to application modeling common problems. It

would, of course, be almost impossible to have a dictionary holding all possible "things" and how they relate to other "things". But the idea could be simplified to become a DDP. In order to find any trace of semantic design in the scripts, a table was created with the following structure:

- Script ID: The name of the SQL script being studied.
- Table Name: The name of the database table.
- Pattern Name: An early name assigned to the pattern. The name is what makes the table "standout" from others. It could be its content (logs, session), its design (tree) or a special attribute (hidden record flag, modified by).

The discovery of DDPs was executed by an automated process by relying on a formal definition. Only thorough analysis of the code made some tables "special", especially when the design is seen repeatedly, in many, un-related applications. The following attributes were considered for candidate patterns:

- Table Content: All the attributes in the table seem to be "special". Many developers seem to model the same table in many applications.
- Table Design: The relationship between this table and other tables, or even its relationship with itself.
- Special Attribute: A special attribute in the table seems to appear in most tables in the same database.

The table content is a typical candidate for detecting a semantic design. Let us illustrate it with an example. A common real world entity is a person. Many applications require the storing of information related to an individual. Such information could be, for example, the first, middle and last name, the phone number, email address, etc. If many of the applications store such information in a similar way, we would have discovered a new DDP (see Table 1).

This exercise presents the following challenges:

- Defining a dictionary of real world attributes and linking each attribute in database table to a word in this dictionary.
- Defining a similarity method for identifying similar structures.
- Defining a process to look into a database and if necessary, expand the dictionary.

The visual analysis of the database designs was rewarding. Quickly, design patterns started to emerge. As Robert Martin [12] highlights the importance of providing pattern descriptions "The revolutionary concept of the GoF book is not the fact that there are patterns; it is the way in which those patterns are documented. Prior to the GoF book, the only good way to learn patterns was to discover them in design documentation, or (more probably) code." There is not a single method of documenting patterns, but since the GoF book is the most referenced work in this domain, this work will follow the same format used by the book.

Table 1 Dictionary of entity modeling.

Field Description	Field Data Type	Linked	Data Item Description
First Name	Sequence of characters	No	A sequence of characters identifying the person's first name.
Last Name	Sequence of characters	No	A sequence of characters identifying the person's last name.
Email Address	Sequence of characters with additional checking for a "@" character and at least a ".".	No	The email address, This field could be unique.
Date of Birth	Date	No	When the person was born
SSN	Sequence of characters	No	Social security number, unique per person, but since the number is never used for computation and is often quite large (11 digits), it is often stored as characters.
Nationality	Foreign key to the Country table.	Yes	The primary nationality of the person.

The study of the database structures for the open source applications yielded to the discovery of the following 24 DDPs:

1. The "Table for Administrative Users" Pattern: Applications that supports login credentials may need to assign administrative privileges to some of the accounts. Instead of adding an administrative attribute to the users table, this pattern suggests to create a clone of the users table where all the administrative accounts are stored. Security is the main driver for this pattern. Write operations on this table are not permitted to the same DBMS users accessing the other tables in the schema.
2. The "Auto Number for Most Tables" Pattern: The "Auto Number" is a feature available in many database engines. It is a special type of attribute, which is assigned an incremental value for every tuple. Because it is unique and never null, it is sometimes used as the primary key in a table. This design pattern recommends the use of this key for all the tables that do not have a primary key.
3. The "Auto Number for All Tables" Pattern: This pattern is an extension to the "Auto Number for Most Tables" pattern, with the difference that it is applied to ALL the tables in the model.
4. The "Created By" Pattern: An attribute to store the id of the user who initially inserted the tuple.
5. The "Created When" Pattern: An attribute to store a timestamp of when the tuple was initially inserted.
6. The "File Outside the Database" Pattern: If a

database model requires the storing of unstructured data, this pattern recommends the storing of the information as a stream of bytes on the file system, and the keeping of a pointer to the file in the database.

7. The “Hidden Records” Pattern: In order to keep deleted records available for auditing, un-deletion or other tracking purposes, it is common to add a flag for each tuple which can mark a record to be logically deleted or hidden.
8. The “Logs” Pattern: The logs table is a tracing and debugging space in a database that is used by an application to store messages in order to allow easy and remote debugging.
9. The “One Table Database” Pattern: This DDP can be used if the information to be persisted is simple to the point where it could be modeled as a single table.
10. The “Record Status” Pattern: When the modeled entity goes through a different set of defined states, this pattern recommends the adding of an attribute as a label for that state. For example, an item could be manufactured, boxed, shipped, retrieved, delivered, etc. This could be a simple method of workflow-enabling an existing application.
11. The “Session” Pattern: A pattern to persist information related to an HTTP session. One of its uses is to store the timestamp of each HTTP request, in order to measure the time between requests. If that time goes beyond a defined limit, the application should mark the HTTP session as expired.
12. The “Software Version” Pattern: This pattern will store the software name, version and patch number in a string stored in table with one row and one column. This will allow any patching script to retrieve the patch level of the application before an eventual execution.
13. The “System Settings” Pattern: A database table that stores a set of attributes with their respective values. It is database version of a JAVA “.properties” file.
14. The “Tree” Pattern: A pattern for modeling a tree-like structure in a database. Each record contains an attribute called “father” storing the primary key of the father node.
15. The “Updated By” Pattern: An attribute to store the id of the user who was the last to modify the tuple.
16. The “Updated When” Pattern: An attribute to store a timestamp of when the tuple was last modified.
17. The “User Group Association” Pattern: This pattern allows the definition of user associations, usually to ease the management of rights and permissions. By associating permissions to groups rather than users, it is possible to set the permissions once to a single group, and assign as many users to the group as needed.
18. The “User Preference” Pattern: It is often required to allow a user to customize an application to his/her taste and needs.
19. The “User and Password” Pattern: Many applications require a user name and a password to identify and authorize the user. In its simplest implementation, a table hosts a user and the hash of the password.
20. The “Workflow” Pattern: A workflow is an application that allows a user to fill a form and then move it through a predefined path, allowing the data to be filled and be reviewed in many stages. There are many open source workflow engines, and many have a particular database design. All the data entered by the user in the form is stored in a single table. Each row is a data item, having attributes linking it to another table to define its data type (integer, float, date, etc.) and item value is stored as binary large object. The value is actually a serialization of a class holding the user input.
21. The “Standard Tables” Pattern: The analysis of all the SQL scripts quickly showed that some typical tables are often present in many system implementations, like MIME types, countries and currencies, ISO codes, time-zones, capitals, international phone extensions, etc.
22. The “Archived Data” Pattern: If a table grows to a point where it hinders the performance of the application accessing it, it is sometimes useful to split it into a “current” and “master”. Both would have the same structure, and the data is periodically moved from the current table to the master
23. The “Schema Software Version” Pattern: This pattern appends the version of the software to the schema name to allow many versions of the same application to run on the same instance of the database.
24. The “Record Locking” Pattern: If the application needs to lock one or more tuples in order to prevent other users from accessing or modifying some data items, this pattern proposes to add a “Locked” flag, a “Locked By” id referencing a user, a “Locked Time” and a “Lock Limit” to automatically release a locked record after a predefined time.

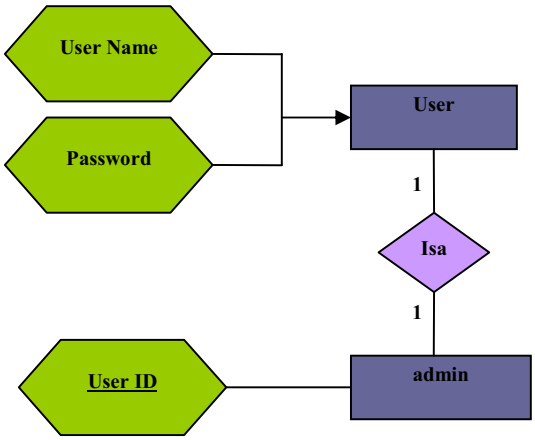
The brief coverage of the discovered DDPs shows the level of design targeted in this paper. Common database design tools can help a developer design complex systems without having to manually write the SQL statements; however, it would difficult to conceive a tool that could assist the database architect in defining the attributes and relationships for modeling a workflow, defining a logging table, or managing software level locking on some tuple. DDPs will expose the experience of other developers in defining systems, and the detailed description of the discovered patterns show that they will be of little help when designing small systems.

We will next elaborate two patterns. Interested readers are referred to [13] for a complete discussion of all the patterns.

The "Table for Administrative Users" Pattern

Some of the applications that authenticate their users by a user name may need to assign administrative privileges to some of the accounts. Since the authority assigned to these persons may be very critical to the software, these accounts are defined in a table, separate from the table holding the usernames:

1. **Pattern Name and Classification:** Table for administrative users.
2. **Intent:** Apply more restrictive database permissions to the table that holds the username of the administrative users.
3. **Also Known As:** Admin Users Table.
4. **Motivation (Forces):** Imagine an application that can allow a user to change the username and password. If the application relies on the database permissions, all users will have read, write and delete access to the users table. One user could then change the username to "admin" or "supervisor" and elevate the permissions [14][15]. In order to prevent this, the privileged users will be created in another table where very few accesses are permitted. The application will only provide additional privileges to the users defined in this table.
5. **Applicability:** Any application with support for user based authentication.
6. **Structure:** The structure of the pattern in UML [16]:



7. **Participants:** The application will validate the username and password first, and if valid, will try to find a record in the admin users table. If match is found the user will be an administrator, and the application should enable the implied functionalities.
8. **Collaboration:** At most, a one to one relationship exists between the users table and the admin users table.
9. **Consequences:** Each credential validation will inquire an additional and often unnecessary read from the administrative table, since not all users are administrators.
10. **Implementation:** Script 16 is an example of such an implementation.
11. **Sample Code:**

```
CREATE TABLE Admin (
Password varchar(20) NOT NULL default '',
Title varchar(128) NOT NULL default '',
Author varchar(128) NOT NULL default '',
Email varchar(128) NOT NULL default '',
Skin varchar(64) default 'standard',
Permissions int(11) unsigned default '0'
) TYPE=MyISAM;
```

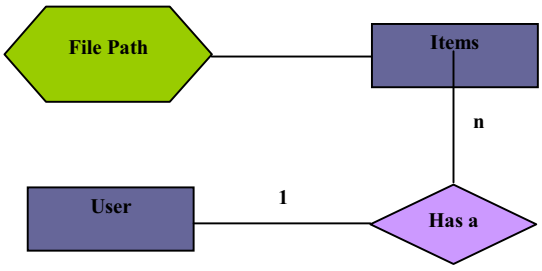
00016.SQL

12. **Known Uses:** The "Table for Administrative Users" pattern was found in all the following applications: 00016.sql.
13. **Related Patterns:** "User and Password".

The "File Outside the Database" Pattern

The increase in demand for document management systems (and similar applications) has caused the creation of this pattern. Applications often need to store meta data about the files being saved. These attributes include the date and time when the file was created or last accessed, a checksum, business related attributes, etc. All this information will be stored in the database, while the file will remain on the file system. However, a path to the file is kept in a character column.

1. **Pattern Name and Classification:** File outside the database.
2. **Intent:** Avoid storing a file in binary large object (BLOB) column.
3. **Also Known As:** Path storing of files.
4. **Motivation (Forces):** An example application would be a document management system that needs to avoid the coding of a file system layer within the application, so instead of storing files and user authorities, only the file path is stored.
5. **Applicability:** Any data centric application running on a system that has support for hierarchical file system.
6. **Structure:** The structure of the pattern in UML:



7. **Participants:** Although all database engines offer the possibility to store BLOBs, some developers seem to prefer to store files outside the database and to conserve only a link.
8. **Collaboration:** The relationship in this DDP, is between a database table and the file system.
9. **Consequences:** Since the access rights on the database

and the file system cannot be enforced to be always in sync, there could be discrepancy between what exists, or what is meant to exist in the application.

10. **Implementation:** This feature is often found in application dealing with many document. This is a simple implementation of a document management system.
11. **Sample Code:**

```
CREATE TABLE muvfs (
   vfs_id       BIGINT NOT NULL,
   vfs_type     SMALLINT NOT NULL,
   vfs_path     VARCHAR(255) NOT NULL,
   vfs_name     VARCHAR(255) NOT NULL,
   vfs_modified  BIGINT NOT NULL,
   vfs_owner    VARCHAR(255) NOT NULL,
   vfs_perms    SMALLINT NOT NULL,
   vfs_data     LONGBLOB,
-- Or, on some DBMS systems:
--  vfs_data     IMAGE,
   PRIMARY KEY   (vfs_id)
);
```

00079.SQL

12. **Known Uses:** The "File Outside the Database" pattern was found in all the following applications: 00012.sql and 00079.sql.
13. **Related Patterns:** None.

To illustrate how a database designer could use a great number of the DDPs covered in a single application, a sample implementation is proposed for the modeling of a bookshop (see Figure 1).

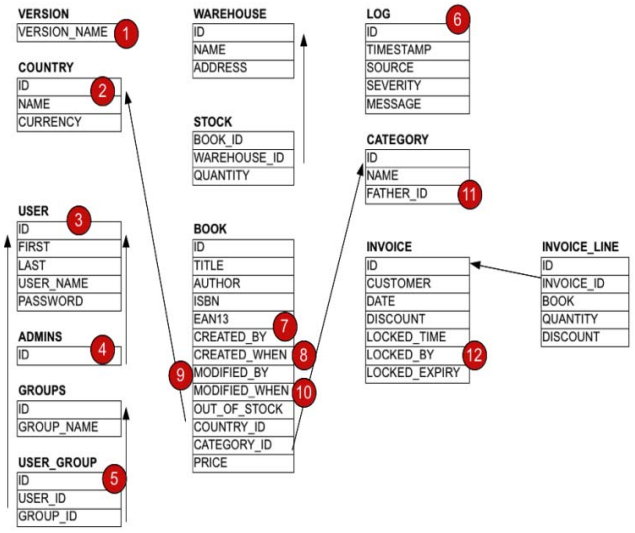


Fig. 1 Sample implementation of a bookshop.

An experienced database designer might see the design in Figure 1 as common or even quite good. But each of the labeled numbers refers to the following database design patterns:

1. Version Name
2. Standard Table
3. Users Table
4. Administrator Table
5. User-group Tables
6. Logs Table
7. Created By
8. Created When
9. Modified By
10. Modified When
11. Tree Table
12. Lock Expiry

The sample example and the nature of the functionality covered by each of the discovered patterns will help categorize them into six distinct sections:

1. System settings – All the DDPs whose implementation has no effect on the features and functionality of the modeled application:
   - The "Software Version" Pattern.
   - The "Schema Software Version" Pattern.
   - The "System Settings" Pattern.
   - The "Logs" Pattern.
2. User, groups and rights management - All the DDPs related to credential management, user permissions, access control and user preferences:
   - The "User Group Association" Pattern.
   - The "Table for Administrative Users" Pattern.
   - The "User And Password" Pattern.
   - The "User Preferences" Pattern.
3. Common settings for all tables in the database – All the DDPs implementing the minimal required attributes for all tables:
   - The "Auto Number for Most Tables" Pattern.
   - The "Auto Number for All Tables" Pattern.
   - The "Created By" Pattern.
   - The "Created When" Pattern.
   - The "Updated By" Pattern.
   - The "Updated When" Pattern.
   - The "Hidden Records" Pattern (Logical delete).
   - The "Archived Data" Pattern (Split a table in two if its size grows beyond a certain threshold).
4. Semantic modeling - Knowing the attributes and relationships of the entities being modeled:
   - The "Standard Tables" Pattern.
5. Workflow proper patterns – All DDPs implementing the required functionality for workflow functionality:
   - The "Record Status" Pattern.
   - The "Workflow" Pattern.
6. A table to handle the web session, should a web interface be developed for accessing the database content:
   - The "Session" Pattern.

It is not uncommon for the database scripts to find half the attributes used to implement patterns, and the other half to model the actual application. If the patterns are self-imposed, what remains to be modeled are the attributes relative to real world applications. These are the attributes that are collected during the software requirements gathering. If the process of collecting these attributes could be automated, it would be possible to build the database model during the early design phase. Unfortunately, the database design is only half the story. More effort is needed to build the relationship between the attributes in the model, define the constraints, the domain limits and the business logic. This would be the ideal implementation. But another approach could be proposed: Partial automation. Sometimes, regular expressions, semantic analysis and advanced statistics can significantly help in attaining an acceptable consistency level. Regular expressions can be used to validate the structure of an attribute. For example, a phone number could be defined as "961-*d*-*ddd ddd*" where each "*d*" is replaced by a digit. Advanced statistics (such as k^{th} order statistics) [17] could be used to detect the upper and lower boundaries for each numerical attribute (bounded data) [18]. Finally, list detection will allow the detection of non-bounded attributes (grouped data).

IV. CONLUSION

Design patterns are of the essence to the software development process. They took the software engineering world by storm. Unfortunately, this was not the case in the realm of databases. Design patterns are a clear need for the database world as they can serve as guiding aids and reference language for designers, especially in their early steps. This paper studied open source, data centric applications and presented 24 database design patterns.

As for future work, more open source projects have to be investigated and more patterns have to be contrived. A balanced organization must be maintained and the profound foundations of why a solution is good must be further investigated using concrete metrics.

ACKNOWLDGEMENT

This work was supported by the Lebanese American University.

REFERENCES

[1] Bourque, P. & Dupuis, R. (1999). Guide to the Software Engineering Body of Knowledge. IEEE Software, November/December, pp. 35 – 44.

[2] Tropashko, A. & Burleson, D.K. (2007). SQL Design Patterns: Expert Guide to SQL Programming, Rampant Techpress.

[3] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). Design Patterns Elements of Reusable Object-oriented Software, Addison Wesley Professional Computing Series, Boston, USA.

[4] Stathopoulou, E., & Vassiliadis, P. (2009). Design Patterns for Relational Databases. ODMG, Technical Report, Retrieved from http://www.odbms.org/download/PP2.PDF.

[5] Vasutiu, V., Vasutiu, F. (2006). Database Design Patterns. Ph.D. Dissertation. University Szeged, Szeged, Hungary.

[6] Vitacolonna, N. (2011). Conceptual Design Patterns for Relational Databases, Proceedings of the 17th International Conference on Information and Software Technologies, Kaunas, Lithuania. pp. 239 – 246.

[7] Beck, K., Crocker, R., Meszaros, G., Vlissides, J., & Coplien, J.O. (1996). Industrial Experience with Design Patterns. Proceedings of the 18th IEEE International Conference on Software Engineering, Washington, DC, USA, pp. 103-114.

[8] Google Code Search (2011). Retrieved on August 28, 2013 from http://code.google.com/apis/codesearch/docs/2.0/reference.html.

[9] www.sourceforge.com. Retrieved on August 28, 2013.

[10] www.github.com. Retrieved on August 28, 2013.

[11] Brown, G. (1993). U.S. National Institute of Standards and Technology, Federal Information Processing Standards Publication. Integration Definition for Information Modeling (IDEF1X 184).

[12] Martin, R.C. (2002). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, USA.

[13] Stephan, G. (2011), Database Design Patterns. Master's Thesis. Lebanese American University, Beirut, Lebanon.

[14] Panda, B., Perrizo, W., Haraty, R. A.: Secure Transaction Management and Query Processing in Multilevel Secure Database Systems. Proceedings of the Symposium on Applied Computing. Phoenix, AZ. (1994)

[15] Haraty, R. A.: C2 Secure Database Management Systems – A Comparative Study. Proceedings of the ACM Symposium on Applied Computing. San Antonio, TX. (1999)

[16] UML Specification version 1.1 - OMG document ad/97-08-11. Retrieved on August 28, 2013 from: http://omg.org.

[17] David, H. A., Nagaraja, H.N. (2003) Order Statistics. John Wiley and Sons, Hoboken, New Jersey, USA.

[18] Sefling, R.J. (1980) Approximation Theorems of Mathematical Statistics. John Wiley and Sons, New York, USA.