DATABASE DESIGN PATTERNS
By
GEORGES BOUTROS STEPHAN

A thesis
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

School of Arts and Sciences
July 2011
LEBANESE AMERICAN UNIVERSITY
School of Arts and Sciences - Beirut Campus

Thesis Proposal Form (Annex I)

Name of Student: Georges Stephan
I.D.#: 199234650

Division: Computer Science and Mathematics

On (dd/mm/yy) June 29, 2007 has presented a Thesis proposal entitled:

Design Patterns for Relational Databases

In the presence of the Committee members and Thesis Advisor:

Ramzi A. Haraty
(Name and Signature of the Thesis Advisor)

Rony Touma
(Name and Signature of Committee Member)

Issam Kouatli
(Name and Signature of Committee Member)

Proposal Approved on (dd/mm/yy): June 29, 2007

Comments/Remarks/Conditions to Proposal:

________________________________________________________________________

Date: 7/8/2007 Acknowledged by [Signature]
(Dean, School of Arts and Sciences)

cc: Chair
Advisor
Student
File Graduate Studies
LEBANESE AMERICAN UNIVERSITY
School of Arts and Sciences - Beirut Campus

Thesis Defense Result Form (Annex II)

Name of Student: GEORGES STEPHAN I.D.#: 199234650

On 22/07/11, has defended a Thesis entitled:

Database Design Patterns

In the presence of the following Committee members:

Advisor: Dr. Ramzi A. Haraty (Name and Signature)

Committee Member: Dr. Rony Touma (Name and Signature)

Committee Member: Dr. Issam Kouatli (Name and Signature)

The student has passed ___ not passed ___ the Thesis defense in partial Fulfillment of the requirements of the degree of MS in Computer Science.

Comments/Required Changes to Thesis due on 22/7/2011

Advisor’s report on completion of above Thesis conditions:

________________________

Changes Approved by Thesis Advisor: R. Haraty Signature:

Date: 22/7/2011

Acknowledged by: [Redacted] (Dean, School of Arts and Sciences)

cc: Registrar
Advisor
File Graduate Studies

Date of the Thesis defense public announcement 8/07/11
LEBANESE AMERICAN UNIVERSITY
School of Arts and Sciences - Beirut Campus

Thesis approval Form (Annex III)

Student Name: Georges Sekle   I.D. #: 19923460
Thesis Title: Database Design Patterns

Program: Masters in Computer Science
Division/Dept: Computer Science & Mathematics
School: School of Arts and Sciences
Approved by: Rony Jouma
Thesis Advisor: [Redacted]
Member: Rony Jouma [Redacted]
Member: Dr. T. Kourli [Redacted]

Date: 22/7/2011

(This document will constitute the first page of the Thesis)
THESIS COPYRIGHT RELEASE FORM

LEBANESE AMERICAN UNIVERSITY NON-EXCLUSIVE DISTRIBUTION LICENSE

By signing and submitting this license, you (the author(s) or copyright owner) grants to Lebanese American University (LAU) the non-exclusive right to reproduce, translate (as defined below), and/or distribute your submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video. You agree that LAU may, without changing the content, translate the submission to any medium or format for the purpose of preservation. You also agree that LAU may keep more than one copy of this submission for purposes of security, backup and preservation. You represent that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone's copyright. If the submission contains material for which you do not hold copyright, you represent that you have obtained the unrestricted permission of the copyright owner to grant LAU the rights required by this license, and that such third-party owned material is clearly identified and acknowledged within the text or content of the submission. IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN LAU, YOU REPRESENT THAT YOU HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT. LAU will clearly identify your name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to your submission.

Name: Georges Stephan

Signature:                                                                 Date: July 22, 2011
PLAGIARISM POLICY COMPLIANCE STATEMENT

I certify that I have read and understood LAU’s Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me. This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Georges Stephan

Signature: 

Date: July 22, 2011
To my family
DATABASE DESIGN PATTERN

Georges Boutros Stephan

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 DP made a significant difference in the delivery speed and product quality. Unfortunately, databases did not yet attract the attention of DP designers. This thesis benefits from the growing popularity of open source, data centric applications, and uncovers 24 DPs related to databases. The relative high number of patterns triggered the creation of a new high level design language, DDRL, (Database Designer Reconnaissance Language), which would allow anyone who knows how to use a word processor to design and populate a normalized database; thus, laying out the blue print for an eventual automation process, and significantly reducing and simplifying all the stages of an upcoming development process. The principles behind DDRL will be elaborated (language grammar, features and output characteristics).

Keywords: DDRL, Development process, JAVA, Database, Design, Design Pattern
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-</td>
<td></td>
</tr>
<tr>
<td><strong>Introduction</strong></td>
<td>1-3</td>
</tr>
<tr>
<td>II-</td>
<td></td>
</tr>
<tr>
<td><strong>Literature Review</strong></td>
<td>4-12</td>
</tr>
<tr>
<td>2.1 - Design Patterns: Elements of Reusable Object-Oriented Software</td>
<td>4</td>
</tr>
<tr>
<td>2.2 – Design Patterns for Relational Databases</td>
<td>5</td>
</tr>
<tr>
<td>2.3 – Database Design Patterns</td>
<td>6</td>
</tr>
<tr>
<td>2.4 – Conceptual Design Patterns for Relational Databases</td>
<td>7</td>
</tr>
<tr>
<td>2.5 - Do Design Patterns Impact Software Quality Positively?</td>
<td>8</td>
</tr>
<tr>
<td>2.6 - Design Patterns for Database Pedagogy – A Proposal</td>
<td>10</td>
</tr>
<tr>
<td>2.7 - Industrial Experience with Design Patterns</td>
<td>11</td>
</tr>
<tr>
<td>III-</td>
<td></td>
</tr>
<tr>
<td><strong>Proposed Solution / Methodology</strong></td>
<td>17-22</td>
</tr>
<tr>
<td>3.1 - Methodology</td>
<td>17</td>
</tr>
<tr>
<td>IV-</td>
<td></td>
</tr>
<tr>
<td><strong>Solution Validation / Findings</strong></td>
<td>23-94</td>
</tr>
<tr>
<td>4.1 - The “Table for Administrative Users” Pattern</td>
<td>24</td>
</tr>
<tr>
<td>4.2 - The “Auto Number for Most Tables” Pattern</td>
<td>26</td>
</tr>
<tr>
<td>4.3 - The “Auto Number for all Tables” Pattern</td>
<td>30</td>
</tr>
<tr>
<td>4.4 - The “Created By” Pattern</td>
<td>32</td>
</tr>
<tr>
<td>4.5 - The “Created When” Pattern</td>
<td>37</td>
</tr>
<tr>
<td>4.6 - The “File Outside the Database” Pattern</td>
<td>40</td>
</tr>
<tr>
<td>4.7 - The “Hidden Records” Pattern</td>
<td>42</td>
</tr>
<tr>
<td>4.8 - The “Logs” Pattern</td>
<td>45</td>
</tr>
<tr>
<td>4.9 - The “One Table Database” Pattern</td>
<td>48</td>
</tr>
<tr>
<td>4.10 - The “Record Status” Pattern</td>
<td>50</td>
</tr>
<tr>
<td>4.11 - The “Session” Pattern</td>
<td>53</td>
</tr>
<tr>
<td>4.12 - The “Software Version” Pattern</td>
<td>55</td>
</tr>
<tr>
<td>4.13 - The “System Settings” Pattern</td>
<td>57</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>4.14 - The “Tree” Pattern</td>
<td>59</td>
</tr>
<tr>
<td>4.15 - The “Updated By” Pattern</td>
<td>61</td>
</tr>
<tr>
<td>4.16 - The “Updated When” Pattern</td>
<td>64</td>
</tr>
<tr>
<td>4.17 - The “User Group Association” Pattern</td>
<td>67</td>
</tr>
<tr>
<td>4.18 - The “User Preferences” Pattern</td>
<td>71</td>
</tr>
<tr>
<td>4.19 - The “User And Password” Pattern</td>
<td>73</td>
</tr>
<tr>
<td>4.20 - The “Workflow” Pattern</td>
<td>76</td>
</tr>
<tr>
<td>4.21 - The “Standard Tables” Pattern</td>
<td>78</td>
</tr>
<tr>
<td>4.22 - The “Archived Data” Pattern</td>
<td>80</td>
</tr>
<tr>
<td>4.23 - The “Schema Software Version” Pattern</td>
<td>83</td>
</tr>
<tr>
<td>4.24 - The “Record Locking” Pattern</td>
<td>86</td>
</tr>
<tr>
<td>4.25 – Sample Implementation</td>
<td>89</td>
</tr>
<tr>
<td>4.26 – Analysis of the Discovered Patterns</td>
<td>91</td>
</tr>
</tbody>
</table>

V- **Conclusions and Recommendations**  95-107

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 - Recommendations</td>
<td>95</td>
</tr>
<tr>
<td>5.2 - The Target Design Characteristics</td>
<td>96</td>
</tr>
<tr>
<td>5.3 - DDRL Stages</td>
<td>97</td>
</tr>
<tr>
<td>5.3.1 - Phase 1 User Identification and Authorization</td>
<td>97</td>
</tr>
<tr>
<td>5.3.2 - Phase 2 Parsing</td>
<td>97</td>
</tr>
<tr>
<td>5.3.3 - Phase 3 Domain and Constrains Definition</td>
<td>99</td>
</tr>
<tr>
<td>5.3.4 - Phase 4 Hinting</td>
<td>100</td>
</tr>
<tr>
<td>5.3.5 - Phase 5 Precision management</td>
<td>101</td>
</tr>
<tr>
<td>5.3.6 - Phase 6 Persisting the Data</td>
<td>101</td>
</tr>
<tr>
<td>5.3.6.1 - Persistence Process Details</td>
<td>101</td>
</tr>
<tr>
<td>5.3.6.2 - Persistence Logging</td>
<td>102</td>
</tr>
<tr>
<td>5.3.7 - Phase 7 Fields Prompting</td>
<td>102</td>
</tr>
<tr>
<td>5.4 - Asynchronous Data Analysis Modules</td>
<td>103</td>
</tr>
<tr>
<td>5.4.1 - Data Limit Detection Module</td>
<td>103</td>
</tr>
<tr>
<td>5.4.2 - Permission Detection Module</td>
<td>103</td>
</tr>
<tr>
<td>5.4.3 - Workflow Detection Module</td>
<td>104</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.4.4 - One to One Relation Detection Module</td>
<td>104</td>
</tr>
<tr>
<td>5.5 - Update and Change Management</td>
<td>104</td>
</tr>
<tr>
<td>5.6 - J-DDRL</td>
<td>106</td>
</tr>
<tr>
<td>5.7 - CONCLUSION</td>
<td>107</td>
</tr>
<tr>
<td>5.8 - Further Work</td>
<td>107</td>
</tr>
<tr>
<td>VI- Bibliography / References</td>
<td>108</td>
</tr>
</tbody>
</table>
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Design Pattern Efficiency</td>
<td>9</td>
</tr>
<tr>
<td>Table 2</td>
<td>The Impact of Design Pattern on Software Quality</td>
<td>10</td>
</tr>
<tr>
<td>Table 3</td>
<td>Semantic Attributes Detection</td>
<td>18</td>
</tr>
<tr>
<td>Table 4</td>
<td>Dictionary of Entity Modeling</td>
<td>20</td>
</tr>
</tbody>
</table>

**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Figure Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>“Table for Administrative Users” in UML</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2</td>
<td>“Auto Number in Most Tables” in UML</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3</td>
<td>“Auto Number for all Tables” in UML</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4</td>
<td>“Created By” in UML</td>
<td>33</td>
</tr>
<tr>
<td>Figure 5</td>
<td>“Created When” in UML</td>
<td>38</td>
</tr>
<tr>
<td>Figure 6</td>
<td>“File Outside the Database” in UML</td>
<td>41</td>
</tr>
<tr>
<td>Figure 7</td>
<td>“Hidden Records” in UML</td>
<td>43</td>
</tr>
<tr>
<td>Figure 8</td>
<td>“Logs” in UML</td>
<td>46</td>
</tr>
<tr>
<td>Figure 9</td>
<td>“One Table Database” in UML</td>
<td>49</td>
</tr>
<tr>
<td>Figure 10</td>
<td>“Record Status” in UML</td>
<td>51</td>
</tr>
<tr>
<td>Figure 11</td>
<td>“Session” in UML</td>
<td>54</td>
</tr>
<tr>
<td>Figure 12</td>
<td>“Software Version” in UML</td>
<td>56</td>
</tr>
<tr>
<td>Figure 13</td>
<td>“System Settings” in UML</td>
<td>58</td>
</tr>
<tr>
<td>Figure 14</td>
<td>“Tree” in UML</td>
<td>60</td>
</tr>
<tr>
<td>Figure 15</td>
<td>“Updated By” in UML</td>
<td>62</td>
</tr>
<tr>
<td>Figure 16</td>
<td>“Updated When” in UML</td>
<td>65</td>
</tr>
<tr>
<td>Figure 17</td>
<td>“User Group Association” in UML</td>
<td>68</td>
</tr>
<tr>
<td>Table</td>
<td>Figure Title</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 18</td>
<td>“User Preferences” in UML</td>
<td>72</td>
</tr>
<tr>
<td>Figure 19</td>
<td>“User and Password” in UML</td>
<td>74</td>
</tr>
<tr>
<td>Figure 20</td>
<td>“Workflow” in UML</td>
<td>77</td>
</tr>
<tr>
<td>Figure 21</td>
<td>“Standard Tables” in UML</td>
<td>79</td>
</tr>
<tr>
<td>Figure 22</td>
<td>“Archived Data” in UML</td>
<td>81</td>
</tr>
<tr>
<td>Figure 23</td>
<td>“Schema Software Version” in UML</td>
<td>84</td>
</tr>
<tr>
<td>Figure 24</td>
<td>“Record Locking” in UML</td>
<td>87</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Flow Diagram of the DDRL Parser</td>
<td>98</td>
</tr>
<tr>
<td>Figure 26</td>
<td>DDRL Input Example</td>
<td>98</td>
</tr>
<tr>
<td>Figure 27</td>
<td>The main tables in the DDRL_MODEL schema</td>
<td>99</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Table Fields Prompting</td>
<td>105</td>
</tr>
<tr>
<td>Figure 29</td>
<td>Adding a New Attribute to a Table</td>
<td>106</td>
</tr>
<tr>
<td>Figure 30</td>
<td>Sample Implementation for a Bookshop</td>
<td>90</td>
</tr>
</tbody>
</table>
CHAPTER ONE
INTRODUCTION

1 - INTRODUCTION

All software development processes, and specially for data centric applications, starts with the gathering and analysis of the user's requirements. Many techniques were proposed, such as the Software Requirements Specification (SRS) document by Bourque, & Dupuis, (2004) , but all approaches are time and resource consuming 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 that triggered the writing of this thesis originated from recent proliferation of open source applications. As many of these projects rely on a persistence engine, the idea was to gather and study the database design of the largest possible number of open source applications. Furthermore, the increasing popularity of design patterns (DP a design pattern a common solution to a common problem found by different people) in many disciplines, and specially in software engineering, was a motivation to try to come up with Database Design Patterns or (DDP).

Till this day, DDP were 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 made it deep in the enterprise. There are many reasons why DDPs are absent from the developers table of tools. First, and as the literature review will confirm, no two authors agreed on the definition of DDPs. Second, the de
facto standard database language (SQL) does not offer the same vocabulary and grammar richness as other programing languages such as JAVA, C++ or Smalltalk; thus, giving the false impression that it offers fewer design possibility, and that therefore, few things could go wrong, specially that relational databases exists for a long time and were extensively documented and that the first, second and third normal forms were formally demonstrated as being the ultimate designs to eliminate redundancy and maximize database performance. Finally, and I believe this to be the major reason for the almost nonexistence of DDPs, is that all the authors writing about them seems to have forgot 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, and theses days, the first place to look for would be in open source repositories, were hundred of thousands of applications exists, and were code is freely shared among projects and repositories.

Pattern discovery may lead to the discovery of anti-patterns. These are patterns should not be used, and will treated and documented just like other patterns. The challenge here is how to determine that the design is bad. In this thesis, and for the sake of simplicity, a DDP will be considered as bad if it open a major security risk, or if it corrupt 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 that the differences. To the point that these differences tends to be subtle. This idea triggered another one, to propose a new programming language, DDRL (Database Designer Reconnaissance Language), which could be used to define these “subtleties”.

What makes DDRL unique compared to other languages such as SQL, is that it is not
intended to be used by an end user, not a programmer. This thesis will first go over the few literature reviews about DDPs, then the DDPs located in the applications that were analyzed will be extensively documented, and finally, the design characteristics of DDRL will be proposed, then the sequence of operations of the interpreter will be explained: User identification, parsing, domain definition, hinting, precision management, persistence and fields prompting. Next, the asynchronous data analysis modules and the change management are explained through examples. Finally, J-DDRL, a open source JAVA reference implementation of DDRL will be illustrated.
The work executed by Gamma et al., is the initial spark that popularized the use of design patterns in the word of computer science, specially 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. The key aspect of software design patterns is that they are discovered. Although a concept might look or could be demonstrated as to be efficient, it could not be categorized as a design patterns if it is not implemented in existing software.

The authors found many patterns and divided them in five categories.

1. Creational patterns : Like the Factory, Lazy initialization and Singleton.
2. Structural patterns : Like Composite, Facade and Proxy.

Finally, the authors encouraged the continuous discovery of design patterns to further enrich the existing list.
2.2 – Design Patterns for Relational Databases by Stathopoulou, & Vassiliadis, (2007)

In this paper, the authors present an approach for interacting with a database. The authors detected three database design patterns, the pivoting, the materialization and the generalization and specialization. The idea proposed is to create a new Facade API that interact with database thru a higher level that is aware of the underlying structure and the relationships that governs it. To change a relationship or to add elements to it, the developer will invoke the API and execute a relationship invocation without using SQL. The authors identified the following 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 attributes types. This pattern is often used by workflow systems, and is referred to as the workflow pattern in this thesis.

- **Materialization**: This pattern allow the implementation of a father child relationship as commonly found in object-oriented languages. The pattern establish 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 for Relational Database**: This pattern implement the 'is a' concept commonly found in object oriented databases. For example, a cedar 'is a' tree, which means that a 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.
The purpose of this paper was purely pedagogical. In order to be efficient, the intermediary API should be able to represent all known types of relationships or all existing design patterns. But students using the API seems to have like the approach and were able to quickly start produce normalized databases.

2.3 – Database Design Patterns by Vasutiu & Vasutiu, (2006)

Of all the reviewed documents, the thesis wrote by Ovidiu and Florina Vasutiu is the closest to the work covered by this document. The differences first is that the patterns are not “discovered”. They are sourced from best practices, and then located in some ERP systems such as SAP.

The work focus on mainly two DPs:

1. Internationalization: Allow an application to provide an interface with support to multiple languages. The concept becomes more challenging when dynamic data (that is data generated a run-time) needs to be available in all supported languages. It is worth to be noted that of all the reviewed scripts, no one seems to provide such functionality.

2. Accessible Transaction Log: All database engines with ACID (atomicity, consistency, isolation, durability) support needs to keep a log of all the executed transactions. The idea is that an image of the data to be changed is kept in this logs, and is retrieved and re-applied in an executed query is rolled back. The information in these logs could be used by any application dealing with a database, but, unfortunately, most database engines do not provide the developers with access to the transaction logs. This is the need that this
pattern is trying to address. This will, for example, allow an application to “undo” a database transaction, even if it was successfully committed.

The detail of the implementation of these patterns, such as the table designs, indexes, read and write operations, are also discussed in details. ERP solutions (specially SAP), are used to illustrate the concept.

2.4 – Conceptual Design Patterns for Relational Databases by Vitacolonna, (2011)

This paper proposes a new approaches to a semi-formal definition of Database Design Patterns (DDPs). Two patterns (Multiple Roles, Hierarchy) are studied. The author proposed an extensions to EER (Extended Entity-Relationship) to represent data modeling specificity such constrains within relationships. But instead of graphically represent EERs, the author proposed the following textual annotation:

- \( A(a_0, a_1, \ldots, a_k) \) denotes an entity type \( A \) with identifier \( a_0 \) and additional attributes \( a_1, \ldots, a_k \).
- \( R(A(m_a, M_a), B(m_b, M_b)) \) denotes a binary relationship type \( R \) that relates entities \( A \) and \( B \); \( A \) (resp., \( B \)) participates to \( R \) with minimum cardinality \( m_a \) and maximum cardinality \( M_a \) (resp., \( m_b \) and \( M_b \));
- \( A(a_0, \ldots, a_k) \rightarrow \{B_1, \ldots, B_l\} \) denotes that \( A \) is a specialization of \( B_1, \ldots, B_l \) with additional attributes
- \( a_0,\ldots,a_k; \) we will write \( A(a_0,\ldots,a_k) \rightarrow B \) when \( A \) has only one parent.
- \( A(a_0,a_1,\ldots,a_k) \Rightarrow \{B_1,\ldots,B_l\} \) denotes that \( A \) is a weak entity type identified by \( B_1,\ldots,B_l \), with (op-
• tional) semi-identifier $a_0$ and with additional attributes $a_1,\ldots,a_k$; we will write $A(a_0,\ldots,a_k) \Rightarrow B$

• when $A$ is identified by a single entity type $B$.

• $(A(m_a,M_a), B(m_b,M_b))$ denotes an aggregation of $A$ and $B$ with cardinalities defined as for relations.

• ships. An aggregation may also appear in the previous notations wherever an entity type occurs.

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

• Multiple Roles: This pattern describes the modeling of a relationship where an entity has multiple relationship types with another. An example could be where all teachers are assigned to a class and some teachers are appointed as dean of schools.

• Hierarchy: This pattern describes the modeling of a hierarchy where data items within the same table are related in a father-child fashion. This pattern will be documented later in this thesis as the “tree” pattern.

In conclusion, the author believes that his semi-formal representation would help document and enrich DDPs, also the “impedance-mismatch” between object-oriented programming languages and relational databases. The author expects to cover new DDPs such as History (covered by DDRL) and Access Control (covered by the “user-password” and “user-group” patterns).

The paper presented by Foutse Khomh presented an interesting aspect of design patterns. The momentum behind the applicability and efficiency of their implementation has lead to a myth that there sole presence in any application is by itself a criteria of quality. The relatively large number of paper and thesis covering or dealing with this technology is an example of their popularity. But are design pattern over hyped? To answer this concern, the authors developed a questionnaire based on the following attributes:

Design Related (Expandability, Simplicity and Reusability).

Implementation Related to implementation (Learnability, Understandability and Modularity).

Runtime Related to runtime (Generality,Modularity at runtime,Scalability and Robustness)

Each quality attribute was evaluated using a six-point Likert scale:

A - Very positive
B – Positive
C - Not significant
D – Negative
E - Very Negative
F - Not applicable

The results of the questionnaire are shown in the table below:

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design Pattern Efficiency</strong></td>
</tr>
</tbody>
</table>

The answers to the survey showed that they sensibly decrease simplicity, learnability, and understandability. Evermore, some patterns, like Flyweight, are considered as impacting most attributes negatively.

Table 2
The Impact of Design Pattern on Software Quality

<table>
<thead>
<tr>
<th>Design Patterns</th>
<th>Expendability (%)</th>
<th>Understandability (%)</th>
<th>Reusability (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>R(%)</td>
<td>E</td>
</tr>
<tr>
<td>A. Factory</td>
<td>+</td>
<td>0.00</td>
<td>+</td>
</tr>
<tr>
<td>Builder</td>
<td>+</td>
<td>0.15</td>
<td>+</td>
</tr>
<tr>
<td>F. Method</td>
<td>+</td>
<td>1.76</td>
<td>-</td>
</tr>
<tr>
<td>Prototype</td>
<td>+</td>
<td>30.36</td>
<td>+</td>
</tr>
<tr>
<td>Singleton</td>
<td>-</td>
<td>0.15</td>
<td>+</td>
</tr>
<tr>
<td>Adapter</td>
<td>+</td>
<td>30.36</td>
<td>-</td>
</tr>
<tr>
<td>Bridge</td>
<td>+</td>
<td>0.37</td>
<td>+</td>
</tr>
<tr>
<td>Composite</td>
<td>+</td>
<td>0.00</td>
<td>-</td>
</tr>
<tr>
<td>Decorator</td>
<td>+</td>
<td>0.15</td>
<td>-</td>
</tr>
<tr>
<td>Facade</td>
<td>+</td>
<td>30.36</td>
<td>+</td>
</tr>
<tr>
<td>Flyweight</td>
<td>-</td>
<td>1.76</td>
<td>-</td>
</tr>
<tr>
<td>Proxy</td>
<td>-</td>
<td>30.36</td>
<td>-</td>
</tr>
<tr>
<td>Ch. Of Resp.</td>
<td>+</td>
<td>0.15</td>
<td>-</td>
</tr>
<tr>
<td>Command</td>
<td>+</td>
<td>5.92</td>
<td>-</td>
</tr>
<tr>
<td>Interpreter</td>
<td>+</td>
<td>5.92</td>
<td>+</td>
</tr>
<tr>
<td>Iterator</td>
<td>+</td>
<td>0.15</td>
<td>+</td>
</tr>
<tr>
<td>Mediator</td>
<td>+</td>
<td>30.36</td>
<td>+</td>
</tr>
<tr>
<td>Memento</td>
<td>-</td>
<td>5.92</td>
<td>-</td>
</tr>
<tr>
<td>Observer</td>
<td>+</td>
<td>0.15</td>
<td>-</td>
</tr>
<tr>
<td>State</td>
<td>+</td>
<td>5.92</td>
<td>+</td>
</tr>
<tr>
<td>Strategy</td>
<td>+</td>
<td>1.76</td>
<td>+</td>
</tr>
<tr>
<td>T. Method</td>
<td>+</td>
<td>0.37</td>
<td>-</td>
</tr>
<tr>
<td>Visitor</td>
<td>+</td>
<td>5.92</td>
<td>-</td>
</tr>
</tbody>
</table>

8 + / 2 − 5 + / 5 − 1 + / 9 −
The study showed that design patterns do not always impact software positively. They are complex structure, that could, if not handled correctly, impact the maintainability and efficiency of an application. Design patterns are just tools, and no developers should be forced to use them.

2.6 - Design Patterns for Database Pedagogy – A Proposal - Marlow, Ku, & Benham, (2005)

The work reported by this paper from Marlowe et al, present the database design patterns from unique and interesting perspective. Some of the design patterns presented by the GoF such as the observer, Momento, and command, could also be applied to database design. Since the design patterns do no specify the implementation at the code level, it is possible to implement them in any programing language. Although the implementor is expected to use object-oriented programing languages, it is possible to realize them in procedural and query languages such as SQL.

In order to efficiently present the idea, the authors produced a UML class diagram representation of the database and it's relationships. Next, the characteristics of the produced diagram are compared to that the diagrams produced by the design patterns reported by the GoF.

Using this technique, the authors showed how modeling table access and query planning is similar to the implementation of the program thru the Iterator, Visitor, and Composite design patterns. Moreover, the authors showed how all database relations could be implemented using the Decorator and Observer patterns.

The authors concludes that since database designs have a mathematical cover (normalization), to validate their efficiency, there seems to be no formal method for designing a
relational database. Common methods are often seen more of 'craftman' work than a scientific work. The recommendation was to use UML to design databases, since it's widely accepted and it's methods cover this aspects and the tools for using it are widely available.

2.7 - Industrial Experience with Design Patterns by Beck, Crocker, Meszaros, Vlissides, & Coplien, (1996)

The authors of this paper analyzed the efficiency of design patterns in large companies such as Hewitt Associates, AT & T, Motorola, BNR and Siemens. The paper includes an separate analysis for each. One of the authors, Kent Beck, is developing a repository for Smalltalk design patterns, which should contain about 90 patterns, 30 of which do not focus on pure programing tasks, but rather testing documentation and scheduling.

• Hewitt Associates : Hewitt Associates had a group of five programers migrating a mainframe application to Smalltalk, and after a being introduced to design patterns, not only did they use them, but started quickly producing very good code, although some of them were new object-oriented programming and Smalltalk.

• Orient Overseas Container Limited (OOCL) : OOCL used an interesting method to propagate the knowledge of design patterns among their 30 developers. Along with one of the authors, they organized bootcamps where programers were requested to find the presence (or confirm the absence) of design patterns in a new large distributed application. The effort was successful, although a resistance to design patterns was observed among experienced programers. To circumvent this problem, the author
recommend that talented developers should be allowed to alter these patterns, which often has the benefit of making them more acceptable.

- AT & T: There seems to be many patterns programs at AT & T. Some of the topics covered are:
  - Fault-tolerant architectures: The authors interviewed all the persons involved in the fault-tolerance, and, as they expected, many patterns emerged. These were then documented and made available to all people at AT & T.
  - Process patterns: Design patterns were identified and implemented to optimize miscellaneous processes within the company, showing that not only software can benefit from design patterns.
  - Object patterns: Object-oriented design patterns were carefully used in new projects as the practitioners of this new technology are expected to learn new design methods, programming languages and architectures. The authors do not recommend adding design pattern to this activity.

- Motorola: The Motorola Cellular Infrastructure Group (CIG) worked on the establishment of the relationship between software architecture and Design Patterns. The initial effort was to achieve a very high degree of software re-usability in order to develop application in one tenth of the time. The challenges at the CIG facing this endeavor were the “strong coupling of OO artifacts” and “super-seeding of short term needs over long term ones”.

One of the main challenges for developing software with less the required time, is to define very clear requirements in order to avoid iterative development. Unfortunately, the CIG faced many communication problem where the used vocabulary in one entity meant
different things in another. This is where DP showed its advantages: By defining the product problems and their solutions through DPs, the CIG come up with the following conclusions:

◦ Design Patterns are so related to object oriented programming languages.
◦ Design Patterns provide an efficient method for communicating solutions to common problems.
◦ Design patterns are difficult and lengthy to write and document, to a point where a tradeoff should be established between the importance of the problem and effort to put in order to document the solutions needed to solve it.
◦ It is difficult to evaluate the efficiency of using patterns to solve a problem.

• BNR : BNR is the research and development subsidiary of NorTel. They started working on a pattern-based approach for product development, before even being exposed to the concept. In this paper, BNR disclosed three categories of patterns:
  ◦ Process Method Patterns : Locating patterns in the requirements.
  ◦ Technical Patterns : Some patterns were discovered in the telephony services, and by naming them and applying those documented by the GoF developers did not have to always provide details of an implementation, since DPs could provide an explanation of how a certain mechanism works.
  ◦ Software Architecture Patterns : Architectural pattern such the Observer or Composite provide a good way of describing and implementing and documenting the operations of a system at a high level.
Just like for Motorola, BNR did not measure any improvement in productivity by using patterns, but it was noticed that communication among developers was improved and the execution of complex design was much improved by using many patterns together.

- Siemens: Siemens organized a department that is responsible for managing design patterns to improve the quality of the software produced by multiple industries. The authors of this part of the paper wanted to study the effectiveness of using patterns for developing applications like the process control of steel mills. The patterns were developed by interviewing two pattern specialists and two domain specialists. The patterns were found to have correctly identified the problem and its solution and that they were a useful representation of the knowledge demanded by their projects. The authors also found that the domain experts are the ideal candidates for writing patterns, which is quite challenging since they usually need to learn and understand and document the patterns, specially that they often have the time to do so. At the end of the project, the pattern documentation was transformed in HTML, in order to present the information in an open format. The authors found that the development of industrial design patterns is similar to that of software, which will make the creation of good control software even easier.

All the works reviewed above, agree that design patterns have a positive impact on software development cycles and very often on product quality. Another interesting aspect is that DPs significantly improve the communication among developers. Unfortunately, database design patterns are only covered for pedagogical purposes. DDPs could be better used. 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.
CHAPTER THREE
PROPOSED SOLUTION METHODOLOGY

3.1 - Methodology

Google offer a search engine called “Google code search”. This search engine crawls all
the open source repositories and search the 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, which were all retained. The next step was to
analyze these scripts to try to find any DDPs. Each of those scripts was renamed with a number
for easy reference.

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. This is so true that by looking at all these designs, it is possible to take
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 (Brown, 1993). The idea is that a
dictionary of real world entities is build, and each entities has all of its attributes and how they
related to others entities already defined. The developers will have to 'pick' an object from the
dictionary and integrate it to 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, 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 spreadsheet table was manually created with the following structure:

<table>
<thead>
<tr>
<th>Script ID</th>
<th>Table Name</th>
<th>Pattern Name</th>
<th>Pattern Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 acos</td>
<td></td>
<td>tree</td>
<td></td>
</tr>
<tr>
<td>3 aros</td>
<td>tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 sessions</td>
<td>session</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 cake_sessions</td>
<td>session</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 uri</td>
<td>session</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 uri</td>
<td>record status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 media_types</td>
<td>std tables</td>
<td>mime types</td>
<td></td>
</tr>
<tr>
<td>8 log_table</td>
<td>logs</td>
<td>Generic – logs messages</td>
<td></td>
</tr>
<tr>
<td>9 system_settings</td>
<td>defaults</td>
<td>a table holding the default values for the system</td>
<td></td>
</tr>
<tr>
<td>10 config</td>
<td>system table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 galleries</td>
<td>tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 galleries</td>
<td>Hidden Record Flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 galleries</td>
<td>created by</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The spreadsheet is organized as follows:

1. **Script ID**: The name of the SQL script being studied.
2. **Table Name**: The name of the database table.
3. **Pattern Name**: An early name assigned to the pattern. The name is what make the table “stand out” from other. It could be its content (logs, session), its design (tree) or a special attribute (Hidden Record Flag, Modified By).
4. **Pattern Parameters**: Any remark, or additional or specificity for the pattern.
The discovery of DDPs was executed by an automated process or by relying on a formal definition. Only thorough analysis of the code made some tables “special”, specially when the design is seen repeatedly, in many, un-related applications. The following attributes were considered for candidates patterns:

- Table Content: All the attributes in the table seem to be special. Many developers seem to model the same table in many application.
- 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's illustrate it with an example. A common real world entity is a person. Many application require the storing of information related to an individual. Such information could be for example his first, middle and last name, his phone number, email address, etc. If many of the application store such information in a similar way, we would have discovered a new DDP.

This exercise present the following challenges:

1. Defining a dictionary of real world attribute, and link each attribute in database table to a world in this dictionary.
2. Define a similarity method for identifying similar structures.
3. Define a process to look into a database and associate and if necessary expand the dictionary.
Table 4
Dictionary of Entity Modeling

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

- **Field Description**: The attribute name.
- **Field Data Type**: The type of data being stored (integer, float, double, date, etc.).
- **Linked**: Is the value of the data item saved in the field, or is it a reference to a value
stored in another table. This is also known as a foreign key.

- **Data Item Description**: The description of the attribute.

The visual analysis of the database designs was rewarding. Quickly, design patterns started to emerge. As Robert Martin (Martin, 2002) 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 Gamma et al. (1995) 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 thesis will follow the same format used by their book. It consist of 13 attributes:

- **Pattern Name and Classification**: A short name of one or two words uniquely describing the pattern.

- **Intent**: A description of the problem that this pattern is supposed to fix.

- **Also Known As**: Other names given for the same pattern

- **Motivation (Forces)**: A scenario consisting of a problem and a context in which this pattern can be used.

- **Applicability**: When it is possible and recommended to use the pattern.

- **Structure**: A UML diagram illustrating the pattern. Since this thesis covers only database patterns, entity relationship (ER) diagrams will be used where possible.

- **Participants**: A description of how the databases, tables and constrains interact with the
application to implement the pattern.

- **Collaboration:** A description of how the databases, tables and constrains interact with each other to implement the pattern.

- **Consequences:** Disadvantages and trade offs of using the pattern.

- **Implementation:** A description of an implementation of this pattern, taken from application using the sample code shown in the next section.

- **Sample Code:** An example of how the pattern can be used with in a selected programming language. The GoF book was targeting object oriented languages, while in this thesis, all the code that will be shown will be in SQL.

- **Known Uses:** The GoF show in this section “examples of real usages of the pattern”. In this section, there will be a list of SQL files were this pattern was discovered.

- **Related Patterns:** A discussion on the relationship between this pattern and other patterns.

Each of the discovered patterns will be documented by covering the 13 points listed above.
The study of the database structures for the open source applications yielded to the discovery of the following DDPs:

1. The “Table for Administrative Users” Pattern.
2. The “Auto Number for Most Tables” Pattern.
3. The “Auto Number for all Tables” Pattern.
4. The “Created By” Pattern.
5. The “Created When” Pattern.
6. The “File Outside the Database” Pattern.
7. The “Hidden Records” Pattern.
8. The “Logs” Pattern.
9. The “One Table Database” Pattern.
10. The “Record Status” Pattern.
11. The “Session” Pattern.
12. The “Software Version” Pattern.
13. The “System Settings” Pattern.
14. The “Tree” Pattern.
15. The “Updated By” Pattern.
16. The “Updated When” Pattern.

17. The “User Group Association” Pattern.

18. The “User Preferences” Pattern.

19. The “User And Password” Pattern.

20. The “Workflow” Pattern.


22. The “Archived Data” Pattern.


24. The “Record Locking” Pattern.

4.1 - 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 separate table, separate from the table holding the usernames.

• **Pattern Name and Classification:** Table for Administrative Users.

• **Intent:** Apply more restrictive database permissions on the table that hold the username of the administrative users.

• **Also Known As:** Admin Users Table.

• **Motivation (Forces):** Imagine an application that can allow a user to change his username and password. If the application relies on the database permissions, all user will have read, write and delete access on the users table. One user could then change his
username to 'admin' or 'supervisor' and elevate his permissions. In order to prevent this, the privileged users will be created in another table a very few accesses are permitted. The application will also only provide extra privileges to the users defined in this table.

- **Applicability:** Any application with support for user based authentication.
- **Structure:** The ER diagram in Figure 1 illustrates the pattern:

![Figure 1. “Table for Administrative Users” in UML. This figure illustrate the pattern.](image-url)

- **Participants:** The application will validate the username and password first, and if valid, will try to find a record in the admin table. If match is found the user will be an administrator, and the application should enable the implied functionalities.
- **Collaboration:** At most, a one to one relationship exists between the users table and the admin table.
- **Consequences:** Each credential validation will inquire an additional and often unnecessary read from the the administrative table, since not all users are admins.
- **Implementation:** The scripts 16 and 69 shown below are examples of such an
implementation.

- **Sample Code:**

```sql
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;
```

```sql
CREATE TABLE vtcal_adminuser (  
  id varchar(50) NOT NULL default '',  
  PRIMARY KEY (id)  
);
```

- **Known Uses:** The “Table for Administrative Users” pattern was found in all the following applications:

  00016.sql, 00069.sql, 00087.sql

- **Related Patterns:** “User and Password”.

### 4.2 - The “Auto Number for Most Tables” Pattern

The “Auto Number” is a feature available in most database engines. It is a special type of columns with is assigned an incremented value for each new records. Since each insert will produce a new number, it is often used as the primary key in a database. This design pattern
recommend the use of this feature for all the tables that do not have any primary key defined.

- **Pattern Name and Classification:** Auto Number for Most Tables.

- **Intent:** Defer the creation of primary key for some table to the database engine.

- **Also Known As:** Serial, Auto Increment in most tables.

- **Motivation (Forces):** One or more tables in a database do not have by design a primary key, then a new integer column is added and defined as a primary key.

- **Applicability:** All data centric applications.

- **Structure:** The ER diagram in Figure 2 illustrates the pattern:

![ER Diagram](image)

*Figure 2.* “Auto Number in Most Tables” in UML. This figure illustrate the pattern.

- **Participants:** Some table in the application have a primary key defined to a single
column, which is automatically assigned by the database engine for each inserted row. The application designer needs not to worry on what makes each data record unique.

- **Collaboration:** The auto number attribute is considered as the foreign key for all the related tables.

- **Consequences:** This design might allow logically duplicate data entries since the unicity is only guaranteed by the primary key. Moreover, if data duplication is a concern, the application will have to issue heavy queries to make sure, before each insert, that the data is truly unique.

- **Implementation:** This technique is heavily used non open source applications, and it is not surprise to find it here.

- **Sample Code:**

```sql
CREATE TABLE MIDAS_Check_Current (  
    TID int NOT NULL,
    HID int NOT NULL,
    CID int NOT NULL,
    Status char(21),
    Response longblob,
    LastCheck DATETIME,
    FirstCheck DATETIME,
    Primary Key (TID, HID, CID)
));
CREATE TABLE MIDAS_Checks (  
    CID int NOT NULL AUTO_INCREMENT,
    Name char(50) NOT NULL,
    Descr char(255),
    Type int,
```
• **Known Uses:** The “Auto Number for Most Tables” pattern was found in all the
following applications:
00040.sql, 00063.sql, 00066.sql, 00088.sql, 0244.sql.

- **Related Patterns:** “Auto Number for all Tables”.

### 4.3 - The “Auto Number for all Tables” Pattern

The “Auto Number” is a feature available in most database engines. It is a special type of columns with is assigned an incremented value for each new records. Since each insert will produce a new number, it is often used as the primary key in a database. This design pattern recommend the the use of this feature for all the tables, even if they have a primary key defined.

- **Pattern Name and Classification:** Auto Number for all Tables.
- **Intent:** Defer the creation of primary key for all table to the database engine.
- **Also Known As:** Serial, Auto Increment in all tables.
- **Motivation (Forces):** No matter what information is contained in any table, whether there are unique data in any row or not, each table will have an “auto number” column and this column will be set as a primary key.
- **Applicability:** All data centric applications.
- **Structure:** The ER diagram in Figure 3 illustrates the pattern:
**Participants:** Every single table in the application has a primary key defined to a single column, which is automatically assigned by the database engine for each inserted row. The application designer needs not to worry on what makes each data record unique.

**Collaboration:** The auto number attribute is considered as the foreign key for all the related tables participating in the relationship.

**Consequences:** The same consequences as the “Primary Key for Some Tables” pattern applies.

**Implementation:** This pattern is much more popular than the “Primary Key for Some Tables”.

**Sample Code:**

```sql
CREATE TABLE IF NOT EXISTS `__CL_COURSE__tool_list` (  
  `id` int(11) NOT NULL auto_increment,  
  `tool_id` int(10) unsigned default NULL,  
  `rank` int(10) unsigned NOT NULL,  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```
CREATE TABLE mantis_bug_relationship_table (  
id int(7) unsigned NOT NULL auto_increment,  
source_bug_id int(7) unsigned NOT NULL default '0',  
destination_bug_id int(7) unsigned NOT NULL default '0',  
relationship_type int(2) NOT NULL default '0',  
PRIMARY KEY (id)  
) TYPE=MyISAM;

**Known Uses:** The “Auto Number for all Tables” pattern was found in all the following applications:

00022.sql, 00023.sql, 28.sql, 29.sql, 00073.sql, 00074.sql, 00081.sql, 00085.sql, 0093.sql, 0245.sql.

**Related Patterns:** “Auto Number for Most Tables”.

### 4.4 - The “Created By” Pattern
When records of an application are accessed by many users, a records references the id of the user who 'owns' or created the data records. Some designs also include the id of the user that last modified the information. This will be covered in details by the “Modified By” pattern.

- **Pattern Name and Classification:** “Created By”.
- **Intent:** Store the ID of the user that created and stored the dataset.
- **Also Known As:** “Added By”
- **Motivation (Forces):** An example would be an invoicing (also known as Point of Sale, POS) application would need to know which users created which invoice.
- **Applicability:** Any data centric application that authenticate users and keep track of their activity.
- **Structure:** The ER diagram in Figure 4 illustrates the pattern:

![Figure 4. “Created By” in UML. This figure illustrates the pattern.](image)

- **Participants:** This DDP will allow the application to keep track of the persons who initially created the data entry. Obviously, the application should implement some sort of
strong user authentication to make this pattern valuable.

- **Collaboration:** Trackable data tables should be linked in a many to one relationship with the users table.

- **Consequences:** In order to be efficient, the records in the users table should never be deleted, or the relationship will break. In companies with a high turnover, the users table might seriously grow, also every item creation will cause a lock on the users table.

- **Implementation:** This pattern is easy to implement and offers minimal user tracking features.

- **Sample Code:**

```sql
CREATE TABLE `arch_users` (  
    `user_id` int(11) NOT NULL auto_increment,  
    `login` varchar(50) NOT NULL default '',  
    `PASSWORD` varchar(50) NOT NULL default '',  
    `real_name` varchar(50) NOT NULL default '',  
    `abilitato` tinyint(4) NOT NULL default '0',  
    `email` varchar(50) NOT NULL default '',  
    `masterkey` tinyint(4) NOT NULL default '0',  
    `classe` int(11) NOT NULL default '0',  
    `chiavi_utente` varchar(100) NOT NULL default '',  
    `n_logon` int(11) NOT NULL default '0',  
    `ultimo_logon` datetime NOT NULL default '0000-00-00 00:00:00',  
    `commento` varchar(200) NOT NULL default '',  
    `token` varchar(50) NOT NULL default '',  
    `sys_dt_creazione` datetime NOT NULL default '0000-00-00 00:00:00',  
    `sys_user_creazione` int(11) NOT NULL default '0',  
    `sys_dt_modifica` datetime NOT NULL default '0000-00-00 00:00:00',  
    `sys_user_modifica` int(11) NOT NULL default '0',
)
```
`sys_optilock` int(11) NOT NULL default '0',
PRIMARY KEY (`user_id`),
KEY `do_logon` (`login`(3), `abilitato`)
) TYPE=MyISAM PACK_KEYS=0 AUTO_INCREMENT=11 ;

CREATE TABLE `arch_classes` (  `id_classe` int(11) NOT NULL auto_increment,  `nome_classe` varchar(15) NOT NULL default '',  `des_classe` varchar(200) NOT NULL default '',  `chiavi` varchar(200) NOT NULL default '',  `sys_dt_creazione` datetime NOT NULL default '0000-00-00 00:00:00',  `sys_user_creazione` int(11) NOT NULL default '0',  `sys_dt_modifica` datetime NOT NULL default '0000-00-00 00:00:00',  `sys_user_modifica` int(11) NOT NULL default '0',  `sys_optilock` int(11) NOT NULL default '0',  PRIMARY KEY (`id_classe`)  ) TYPE=MyISAM AUTO_INCREMENT=7 ;
```

CREATE TABLE users (  uid mediumint(8) unsigned NOT NULL auto_increment,  name varchar(60) NOT NULL default '',  uname varchar(25) NOT NULL default '',  email varchar(60) NOT NULL default '',  url varchar(100) NOT NULL default '',  user_avatar varchar(30) NOT NULL default 'blank.gif',  user_regdate int(10) unsigned NOT NULL default '0',  user_icq varchar(15) NOT NULL default '',  user_from varchar(100) NOT NULL default '',  user_sig tinytext NOT NULL,  user_viewemail tinyint(1) unsigned NOT NULL default '0',
actkey varchar(8) NOT NULL default '',
user_aim varchar(18) NOT NULL default '',
user_yim varchar(25) NOT NULL default '',
user_msnm varchar(100) NOT NULL default '',
pass varchar(32) NOT NULL default '',
posts mediumint(8) unsigned NOT NULL default '0',
attachsig tinyint(1) unsigned NOT NULL default '0',
rank smallint(5) unsigned NOT NULL default '0',
level tinyint(3) unsigned NOT NULL default '1',
theme varchar(100) NOT NULL default '',
timezone_offset float(3,1) NOT NULL default '0.0',
last_login int(10) unsigned NOT NULL default '0',
umode varchar(10) NOT NULL default '',
uorder tinyint(1) unsigned NOT NULL default '0',
notify_method tinyint(1) NOT NULL default '1',
notify_mode tinyint(1) NOT NULL default '0',
user_occ varchar(100) NOT NULL default '',
bio tinytext NOT NULL,
user_intrest varchar(150) NOT NULL default '',
user_mailok tinyint(1) unsigned NOT NULL default '1',
PRIMARY KEY  (uid),
KEY uname (uname),
KEY email (email),
KEY uiduname (uid,uname),
KEY unamepass (uname,pass)
) TYPE=MyISAM;

CREATE TABLE image (  
    image_id mediumint(8) unsigned NOT NULL auto_increment,
    image_name varchar(30) NOT NULL default '',
    image_nicename varchar(255) NOT NULL default '',
    image_mimetype varchar(30) NOT NULL default '',
)
• **Known Uses:** The “Created By” pattern was found in all the following applications:

00010.sql, 00058.sql, 00059.sql, 00063.sql, 00093.sql.

• **Related Patterns:** “Created When”, “User and Password”.

### 4.5 - The “Created When” Pattern

It is often important to know when a certain data record was created. This information is often used by reporting and auditing applications.

• **Pattern Name and Classification:** Created When.

• **Intent:** Keep track of the date and time when the record was created.

• **Also Known As:** Creation timestamp.

• **Motivation (Forces):** An example would be a accounting application that would need to generate a report for all the transactions that occurred on a certain day. Each transaction will be stored as a record in a table, with a timestamp associated to each.
• **Applicability:** Any data centric application.

• **Structure:** The ER diagram in Figure 5 illustrates the pattern:

![Created When - Items](image)

*Figure 5.* “Created When” in UML. This figure illustrates the pattern.

• **Participants:** If implemented within the database engine, this feature is completely independent from the application. But many database engines do not allow the creation of more than one timestamp per data record. If that timestamp is already used, then the implementation of this pattern might require application changes.

• **Collaboration:** There are no relationships between table to implement this pattern.

• **Consequences:** If no more than one time stamp is allowed per data record, using it for this pattern will prevent it's uses in the future. The timestamp would have been consumed.

• **Implementation:** This pattern is often used in conjunction with the “Created By” pattern.

• **Sample Code:**

```sql
CREATE TABLE IF NOT EXISTS document (  
docid INT(9) NOT NULL AUTO_INCREMENT,  
objtype TINYINT UNSIGNED NOT NULL DEFAULT 0,  
parent INT(9) NOT NULL DEFAULT 0,  
title VARCHAR(255) NOT NULL DEFAULT '' ,  
created DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',  
hits INT(9) DEFAULT NULL,  
identifier VARCHAR(255) DEFAULT NULL,  
description TEXT NOT NULL,
```
• **Known Uses:** The “Created When” pattern was found in all the following applications:

00010.sql, 00038.sql, 00049.sql, 00058.sql, 00068.sql, 00074.sql, 00085.sql, 00086.sql, 00087.sql, 00088.sql, 00093.sql.
4.6 - The “File Outside the Database” Pattern

There are two ways for a data-centric application to store files. One is to store in a table as a BLOB (Binary Large OBject) column, another is to store in the file system and keep its path in a text column. This pattern covers the second case.

- **Pattern Name and Classification**: File outside the database.
- **Intent**: Avoid storing a file in a binary large object (BLOB) column.
- **Also Known As**: Path storing of files.
- **Motivation (Forces)**: An example application would be a document management system that want to avoid the coding of a file system layer within the application, so instead of storing files, paths, and user authorities, only the file path is stored.
- **Applicability**: Any data-centric application running on a system that has support for hierarchical file system.
- **Structure**: The ER diagram in Figure 6 illustrates the pattern:
• **Participants:** Although all database engines offer the possibility to store BLOB (binary large objects), some developers seem to prefer to store files outside the database and to conserve only a link.

• **Collaboration:** The relationship in this DDP, is between a database table and the file system.

• **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 exist, or what is meant to exist in the application.

• **Implementation:** This feature is often found in application dealing with a lot of document. This is a simple implementation of a document management system.

• **Sample Code:**

```sql
CREATE TABLE entries (  
id int(11) NOT NULL auto_increment,  
image varchar(200) default NULL,  
file varchar(200) NOT NULL,
```

---

Figure 6. “File Outside the Database” in UML. This figure illustrates the pattern.
• **Known Uses:** The “File Outside the Database” pattern was found in all the following applications:

00012.sql, 00079.sql.

• **Related Patterns:** None.

### 4.7 - The “Hidden Records” Pattern

In order to keep deleted records available for auditing and un-deletion, it is common to
add a flag to each record which can mark a record a logically deleted or hidden.

- **Pattern Name and Classification:** Hidden Records.

- **Intent:** Hide some records from accessing applications.

- **Also Known As:** Logical Delete.

- **Motivation (Forces):** An example would be an application that would need to delete some records from all forms and reports while keeping them in the database for reporting and un-deletion.

- **Applicability:** Any data centric application.

- **Structure:** The ER diagram in Figure 7 illustrates the pattern:

![Figure 7. “Hidden Records” in UML. This figure illustrates the pattern.](image)

- **Participants:** The application always filters the data it is reading by specifying that all records should be visible.

- **Collaboration:** All relationships are preserved in this scheme, since no data is deleted.

- **Consequences:** Deleted data is always present in the tables, and cannot be moved, thus significantly increasing the size in scenarios where a lot of data items often tagged as hidden.

- **Implementation:** Often implemented where data tracking is needed for deleted / hidden
information.

- **Sample Code:**

```sql
CREATE TABLE `posts` (
    `post_id` int(10) unsigned NOT NULL auto_increment,
    `modification_time` timestamp NOT NULL default CURRENT_TIMESTAMP
        COMMENT 'time of last edit',
    `creation_time` timestamp NULL default NULL COMMENT 'time when row was first inserted',
    `user_id` int(10) unsigned NOT NULL COMMENT 'authorization identifier',
    `topic_id` int(10) unsigned NOT NULL COMMENT 'uniquely identifies forum topic',
    `is_visible` tinyint(3) unsigned NOT NULL default '1' COMMENT 'Is the post visible to anonymous users?',
    `subject` varchar(128) collate utf8_unicode_ci NOT NULL COMMENT 'subject of this post',
    `content` text collate utf8_unicode_ci NOT NULL COMMENT 'body of the forum post',
    PRIMARY KEY (`post_id`),
    KEY `user_id` (`user_id`),
    KEY `visible` (`is_visible`)
) ENGINE=MyISAM AUTO_INCREMENT=20 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci COMMENT='List of all forum posts';
```

```sql
CREATE TABLE books ( id int(11) NOT NULL auto_increment, invisible int(1) default 0, author text NOT NULL, book text NOT NULL, cat int(11), in_ text, in_extra text, location text,
```
**Known Uses:** The “Hidden Record” pattern was found in all the following applications:

00010.sql, 00029.sql, 0277.sql.

**Related Patterns:** “Record Status”.

**4.8 - The “Logs” Pattern**

The logs table is a tracing and debug space in a database that is used by the application to store messages to enable easy and remote debugging. The idea is that a table store a “message”, or a long text column, a source representing the module that generated the message and a timestamp when the record was generated. More advanced design could include a severity level, and even a action id, hinting what could be done to treat an eventual problem.

**Pattern Name and Classification:** Logs

**Intent:** Provide a tracing and debugging space.

**Also Known As:** Tracing
• **Motivation (Forces):** An example would be a logger API, like log4j which provide the programmer with an interface (in object oriented terms), to implement “writers” to handle the output. Such writers could then write to a table, making it easy to check the status of an application thru any SQL tool, and even make statistical reports on the status of the running application.

• **Applicability:** Any application. Some operating systems like Guardian (HP's non stop servers) implement such pattern.

• **Structure:** The ER diagram in Figure 8 illustrates the pattern:

![Figure 8](image)

*Figure 8. “Logs” in UML. This pattern illustrates the pattern.*

• **Participants:** The inclusion of logging functionalities in Java since version 1.5, is one of many signs of the importance of logging in modern applications. APIs like Java logging and log4j, allow the definition of a DB logger, which automatically writes all logged messages to a database table. It is also not uncommon for developers to write their own logging APIs, specially if the platform they use does not offer such a functionality.

• **Collaboration:** The logging table is not related to other table in the database.

• **Consequences:** If implemented synchronously, the logging will slow the application since each logging requires often, unnecessary I/O. If done asynchronously, the logging might not be very accurate since the logging flow and the application flow could have
happened in a different sequence than that showed by the logs.

- **Implementation**: Useful when real-time, remote logging is required.

- **Sample Code**:

```sql
CREATE TABLE Log (  
    LogId INTEGER,  
    JobId INTEGER UNSIGNED REFERENCES Job NOT NULL,  
    Time DATETIME NOT NULL,  
    LogText TEXT NOT NULL,  
    PRIMARY KEY(LogId)  
);  
```

```sql
CREATE TABLE `task_log` (  
    `task_log_id` INT(11) NOT NULL auto_increment,  
    `task_log_task` INT(11) NOT NULL default '0',  
    `task_log_name` VARCHAR(255) default NULL,  
    `task_log_description` TEXT,  
    `task_log_creator` INT(11) NOT NULL default '0',  
    `task_log_hours` FLOAT DEFAULT "0" NOT NULL,  
    `task_log_date` DATETIME,  
    `task_log_costcode` VARCHAR(8) NOT NULL default '',  
    `task_log_problem` TINYINT( 1 ) DEFAULT '0',  
    `task_log_reference` TINYINT( 4 ) DEFAULT '0',  
    `task_log_related_url` VARCHAR( 255 ) DEFAULT NULL,  
    PRIMARY KEY (`task_log_id`),  
    KEY `idx_log_task` (`task_log_task`)  
);  
```
- **Known Uses:** The “Logs” pattern was found in all the following applications:
  00058.sql, 00072.sql, 00073.sql, 00076.sql, 00079.sql, 00085.sql, 00089.sql, 00093.sql,
  00099.sql, 00179.sql, 00187.sql, 00188.sql, 00191.sql, 00209.sql, 00212.sql, 00218.sql,
  00221.sql, 00239.sql, 00246.sql, 00252.sql, 00278.sql, 00280.sql, 00282.sql, 00283.sql,
  00284.sql.

- **Related Patterns:** None.

### 4.9 - The “One Table Database” Pattern

Sometimes an application might need to persist and retrieve a large but simple set of
information that could be modeled in a single table.

- **Pattern Name and Classification:** One Table Database.
- **Intent:** Use a database to store simple data sets.
- **Also Known As:** Single Table Applications.
- **Motivation (Forces):** An example is a small application that has support for user and
  passwords. All other information handled by the application are not persisted on disk.
  This application will rely on a single table database to store the credentials.
- **Applicability:** Any application with support to database access.
- **Structure:** The ER diagram in Figure 9 illustrates the pattern:
Figure 9. “One Table Database” in UML. This figure illustrates the pattern.

- **Participants:** When simple data needs to be persisted, this DDP seems to be ideal. Specially if the application is planned to grow into a data-centric one.

- **Collaboration:** Since there is only one table, there are no relationships with other entities.

- **Consequences:** If simple data persistence is required, a relational database engine with all it's overhead might be a killer.

- **Implementation:** Typical to small application with aspirations.

- **Sample Code:**

```sql
CREATE TABLE `score` (  
`id` int(10) unsigned NOT NULL auto_increment,  
`date_created` timestamp NOT NULL default '0000-00-00 00:00:00' on update CURRENT_TIMESTAMP,  
`username` varchar(64) NOT NULL default '',  
`score` int(11) NOT NULL default '0',  
`description` text NOT NULL,  
`soft_ver` varchar(40) default NULL,  
PRIMARY KEY (`id`),  
KEY `score` (`score`)
```
• **Known Uses:** The “One Table Database” pattern was found in all the following applications:

  00025.sql, 00031.sql, 00033.sql, 00035.sql, 00043.sql, 00057.sql, 00075.sql, 00086.sql.

• **Related Patterns:** None.

### 4.10 - The “Record Status” Pattern

Sometimes it is required to make a dataset pass by many “states”. It is different than the logical delete, in that a record can pass by many logical stages, none of which is to mark it as deleted.

• **Pattern Name and Classification:** Record Status
• **Intent:** Tag records

• **Also Known As:** Stateful records

• **Motivation (Forces):** An application can simulate a workflow by tagging each records with a “status”, while each group of users filter the available records by the status he is allowed to access.

• **Applicability:** Any data centric application.

• **Structure:** The ER diagram in Figure 10 illustrates the pattern:

![ER Diagram](image-url)

*Figure 10. “Record Status” in UML. This pattern illustrates the pattern.*

• **Participants:** Each record is assigned a status to mark the data attribute go through phases. The application, should know with record to pick when reading and how to set the status when saving.

• **Collaboration:** A simple attribute, which is not linked to another table.

• **Consequences:** There is no finite set of statuses.

• **Implementation:** Simplest implementation of a workflow.

• **Sample Code:**

```sql
create table server (  
    id int auto_increment primary key,  
    scheme varchar(10) not null default "http",  
    host varchar(100) not null,  
);```
CREATE TABLE xoopscomments (  
    com_id mediumint(8) unsigned NOT NULL auto_increment,  
    com_pid mediumint(8) unsigned NOT NULL default '0',  
    com_rootid mediumint(8) unsigned NOT NULL default '0',  
    com_modid smallint(5) unsigned NOT NULL default '0',  
    com_itemid mediumint(8) unsigned NOT NULL default '0',  
    com_icon varchar(25) NOT NULL default '',  
    com_created int(10) unsigned NOT NULL default '0',  
    com_modified int(10) unsigned NOT NULL default '0',  
    com_uid mediumint(8) unsigned NOT NULL default '0',  
    com_ip varchar(15) NOT NULL default '',  
    com_title varchar(255) NOT NULL default '',  
    com_text text NOT NULL,  
    com_sig tinyint(1) unsigned NOT NULL default '0',  
    com_status tinyint(1) unsigned NOT NULL default '0',  
    com_exparams varchar(255) NOT NULL default '',  
    dohtml tinyint(1) unsigned NOT NULL default '0',  
    dosmiley tinyint(1) unsigned NOT NULL default '0',  
    doxcode tinyint(1) unsigned NOT NULL default '0',  
    doimage tinyint(1) unsigned NOT NULL default '0',  
    dobr tinyint(1) unsigned NOT NULL default '0',  
    PRIMARY KEY (com_id),  
    KEY com_pid (com_pid),  
    KEY com_itemid (com_itemid),
)
• **Known Uses:** The “Record Status” pattern was found in all the following applications: 00007.sql, 00010.sql, 00058.sql, 00059.sql, 00063.sql, 00074.sql, 00093.sql.

• **Related Patterns:** “Hidden Records”

### 4.11 - The “Session” Pattern

This pattern is implemented to compensate the fact that the HTTP protocol used to access all web applications is stateless. That is, there is no context. Each request should bring all the needed variables that needs to be processed.

• **Pattern Name and Classification:** Session

• **Intent:** Store information about a logical HTTP session.

• **Also Known As:** Web Sessions.

• **Motivation (Forces):** An example application would be an application that would automatically log off a user from a web site after a certain time of inactivity. Each time the user interact with the system, his cookie is updated on the back end database. If the last time it was updated is more than the session timeout, the application will delete the cookie from the session table and redirect him to the login page.
• **Applicability:** Any web application with a database backend.

• **Structure:** The ER diagram in Figure 11 illustrates the pattern:

![ER Diagram](image)

*Figure 11. “Session” in UML. This figure illustrates the pattern.*

• **Participants:** Each HTTP request (ie HTTP GET or HTTP POST) is validated against the session table. If expired, the record is deleted. If not found, the web application responds with a “Session Timeout” error message.

• **Collaboration:** This DDP is implemented in a single table with no relationships with other tables.

• **Consequences:** Each invocation in each page should check the session table.

• **Implementation:** All web applications with authentication and in need to store variables per session.

• **Sample Code:**

```sql
CREATE TABLE horde_sessionhandler (  
    session_id VARCHAR(32) NOT NULL,  
    session_lastmodified INT NOT NULL,  
    session_data LONGBLOB,  
    PRIMARY KEY (session_id)  
);  
-- Or, on some DBMS systems:  
-- session_data IMAGE,
```

54
CREATE TABLE CPG_sessions (  
session_id varchar(40) NOT NULL default '',  
user_id int(11) default '0',  
time int(11) default NULL,  
remember int(1) default '0',  
PRIMARY KEY (session_id)  
) TYPE=MyISAM COMMENT='Used to store sessions';

• **Known Uses:** The “Session” pattern was found in all the following applications:
  
  00002.sql, 00005.sql, 0007.sql, 00010.sql, 00020.sql, 00040.sql, 00057.sql, 00061.sql,  
  00062.sql, 00069.sql, 00076.sql, 0244.sql.

• **Related Patterns:** None.

### 4.12 - The “Software Version” Pattern

This pattern will store the software name, version and patch level in string stored in table  
with one row and one column. This will allow any patching to check what is installed.

• **Pattern Name and Classification:** Software Version

• **Intent:** Store the software version in database.

• **Also Known As:** Application patch level.
• **Motivation (Forces):** An example would be an application that would want to prevent users from downgrading to an older patch or version. Before installation, the patch will check the software version, and exit if the version is not compatible.

• **Applicability:** Any data centric application.

• **Structure:** The ER diagram in Figure 12 illustrates the pattern:

![Software Version - Version](image)

*Figure 12.* “Software Version” in UML. This figure illustrate the pattern.

• **Participants:** A table containing a single row store the version of the software.

• **Collaboration:** During installation, or during startup, the application can look up in this table to see if the database design needs to be updated.

• **Consequences:** If, for some reasons, this table is illegally modified, and if the application modify the database on startup, the installation script could be re-executed, which could corrupt the entire database.

• **Implementation:** Application that would automate application update.

• **Sample Code:**

```sql
CREATE TABLE `schema_info` (  `version` int(11) default NULL ) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

```sql
CREATE TABLE `gammu` (  
```
```
`Version` tinyint(4) NOT NULL default '0'
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

• **Known Uses:** The “Software Version” pattern was found in all the following applications:
  00010.sql, 00032.sql, 00058.sql, 00074.sql, 00088.sql, 00093.sql, 00279.sql.

• **Related Patterns:** “Schema SoftwareVersion”.

---

### 4.13 - The “System Settings” Pattern

This pattern is for a database table that store using a tag – value scheme (just like a “.properties” file in Java, in a database.

• **Pattern Name and Classification:** System Settings

• **Intent:** Store global, application wide settings.

• **Also Known As:** System configuration.

• **Motivation (Forces):** The most common example is for a web based application that needs to store information about the connectivity to the database (host name, port, user credentials), but also system wide settings like working and temporary directories, etc.).

• **Applicability:** Any Application.

• **Structure:** The ER diagram in Figure 13 illustrates the pattern:
• **Participants:** A single table with two attributes. The application checks this table every time a system parameter needs to be checked.

• **Collaboration:** The table is not connect to any other table.

• **Consequences:** The application might need to cache these values or else each checking of a system parameter will inquire I/O to the database.

• **Implementation:** Applications in need of dynamic and remote application parametrization.

• **Sample Code:**

```sql
CREATE TABLE defaults (  
id serial primary key,
modified_date date default CURRENT_DATE,
modified_date_function date default now(),
fixed_date date default '2004-01-01',
modified_time timestamp default CURRENT_TIMESTAMP,
modified_time_function timestamp default now(),
fixed_time timestamp default '2004-01-01 00:00:00.000000-00',
char1 char(1) default 'Y',
char2 character varying(50) default 'a varchar field',
char3 text default 'a text field',
positive_integer integer default 1,
negative_integer integer default -1,
decimal_number decimal(3,2) default 2.78,
multiline_default text DEFAULT E'--- []

'::text
```

Figure 13. “System Settings” in UML. This figure illustrates the pattern.
CREATE TABLE bayes_global_vars (  
  variable varchar(30) NOT NULL default '',  
  value varchar(200) NOT NULL default '',  
  PRIMARY KEY (variable)  
) TYPE=MyISAM;

INSERT INTO bayes_global_vars VALUES ('VERSION','3');

- **Known Uses:** The “System Settings” pattern was found in all the following applications:
  00009.sql, 00010.sql, 00020.sql, 00022.sql, 00030.sql.

- **Related Patterns:** None.

### 4.14 - The “Tree” Pattern

When a tree-like structure needs to be modeled in a database, such as, for example, a family tree, each record will contain a unique key and another column called 'father' storing the key of the father node.

- **Pattern Name and Classification:** Tree

- **Intent:** Some a hierarchical data structure in a database table.

- **Also Known As:** Tree
• **Motivation (Forces):** The most common example of a tree is an application that needs to model a family tree relationship. The tree will be modeled as a persons table, where each row represents a set of information about a person (first, middle, and last name) and will also have a key. Since each person in existence has a father and a mother, each row will have a foreign key pointing to the id of the mother, and another for the father.

• **Applicability:** Any application that needs to model a hierarchical set of information.

• **Structure:** The ER diagram in Figure 14 illustrates the pattern:

![ER Diagram](image)

*Figure 14. “Tree” in UML. This figure illustrates the pattern.*

• **Participants:** The application can render a hierarchical view of data items.

• **Collaboration:** Other tables are connected through the parent key only.

• **Consequences:** The query to retrieve and fill data is complex.

• **Implementation:** Application that needs to model a hierarchy.

• **Sample Code:**

```sql
CREATE TABLE acos (
    id INTEGER(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    parent_id INTEGER(10) DEFAULT NULL,
```
4.15 - The “Updated By” Pattern

Given a table of users, and a set of records, this pattern recommend to save the id of each user that modify an data item.
- **Pattern Name and Classification:** Updated By

- **Intent:** Keep a record of the user that last updated the data item.

- **Also Known As:** Last Updated By

- **Motivation (Forces):** An application for loan processing sends a set of loan to a group of credit officers. It is very important for the credit manager to know which officer approved which loan, in case a review on the case is needed.

- **Applicability:** Any application that needs to keep track of who last updated critical data.

- **Structure:** The ER diagram in Figure 15 illustrates the pattern:

```
Figure 15. “Updated By” in UML. This figure illustrates the pattern.
```

- **Participants:** This DDP will allow the application to keep track of the persons who last modified the data entry. Obviously, the application should implement some sort of strong user authentication to make this pattern valuable.

- **Collaboration:** Trackable data tables should be linked in a many to one relationship with the users table.
• **Consequences:** In order to be efficient, the records in the users table should never be deleted, or the relationship will break. In companies with a high turn over, the users table might seriously grow, also every item creation will cause a lock on the users table.

• **Implementation:** This pattern is easy to implement and offers minimal user tracking features.

• **Sample Code:**

```sql
CREATE TABLE AD_Attachment(
    AD_Attachment_ID NUMBER(10, 0) NOT NULL,
    AD_Client_ID NUMBER(10, 0) NOT NULL,
    AD_Org_ID NUMBER(10, 0) NOT NULL,
    IsActive CHAR(1) DEFAULT 'Y' NOT NULL,
    Created DATE DEFAULT SysDate NOT NULL,
    CreatedBy NUMBER(10, 0) NOT NULL,
    Updated DATE DEFAULT SysDate NOT NULL,
    UpdatedBy NUMBER(10, 0) NOT NULL,
    AD_Table_ID NUMBER(10, 0) NOT NULL,
    Record_ID NUMBER(10, 0) NOT NULL,
    Title VARCHAR2(60) NOT NULL,
    Text VARCHAR2(2000),
    BinaryData BLOB,
    CHECK (IsActive in ('Y','N')),
    CONSTRAINT AD_Attachment_Key PRIMARY KEY (AD_Attachment_ID)
)
```

```sql
CREATE TABLE /*$wgDBprefix*/revision (  
    rev_id int(8) unsigned NOT NULL auto_increment,
    rev_page int(8) unsigned NOT NULL,
```

63
```
rev_text_id int(8) unsigned NOT NULL,
rev_comment tinyblob NOT NULL default '',
rev_user int(5) unsigned NOT NULL default '0',
rev_user_text varchar(255) binary NOT NULL default '',
rev_timestamp char(14) binary NOT NULL default '',
rev_minor_edit tinyint(1) unsigned NOT NULL default '0',
rev_deleted tinyint(1) unsigned NOT NULL default '0',

PRIMARY KEY rev_page_id (rev_page, rev_id),
UNIQUE INDEX rev_id (rev_id),
INDEX rev_timestamp (rev_timestamp),
INDEX page_timestamp (rev_page, rev_timestamp),
INDEX user_timestamp (rev_user, rev_timestamp),
INDEX usertext_timestamp (rev_user_text, rev_timestamp)
)
```
• **Pattern Name and Classification:** Updated When

• **Intent:** Detect when the data record was last modified.

• **Also Known As:** Record Timestamp.

• **Motivation (Forces):** Imagine a ticket reservation application. Using this pattern, it is possible to know when each ticket was reserved. If the user performing the reservation is taking too much time to register the request, another user might “steal” his ticket. When saving it back the original user will detect that the record timestamp was changed and will report to the user that the ticket is no longer available.

• **Applicability:** Any data centric application. Some database engines provide this function by default.

• **Structure:** The ER diagram in Figure 16 illustrates the pattern:

```
+----------------+              +----------------+              +----------------+
| Updated When   |              | Items           |              | Updated When   |
+----------------+              +----------------+              +----------------+
```

*Figure 16. “Updated When” in UML. This figure illustrates the pattern.*

• **Participants:** If implemented within the database engine, this feature is completely independent from the application. But many database engines do not allow the creation of more than one timestamp per data record. If that timestamp is already used, then the implementation of this pattern might require application changes.

• **Collaboration:** There are no relationships between table to implement this pattern.

• **Consequences:** If no more than one time stamp is allowed per data record, using it for this pattern will prevent it's uses in the future. The timestamp would have been consumed.
• **Implementation:** This pattern is often used in conjunction with the “Updated By” pattern.

• **Sample Code:**

```sql
CREATE TABLE store (  
    store_id TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,  
    manager_staff_id TINYINT UNSIGNED NOT NULL,  
    address_id SMALLINT UNSIGNED NOT NULL,  
    last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,  
    PRIMARY KEY  (store_id),  
    UNIQUE KEY idx_unique_manager (manager_staff_id),  
    KEY idx_fk_address_id (address_id),  
    CONSTRAINT fk_store_staff FOREIGN KEY (manager_staff_id) REFERENCES staff (staff_id) ON DELETE RESTRICT ON UPDATE CASCADE,  
    CONSTRAINT fk_store_address FOREIGN KEY (address_id) REFERENCES address (address_id) ON DELETE RESTRICT ON UPDATE CASCADE  
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```sql
CREATE TABLE `agenti_noti` (  
    `id_agente` int(11) NOT NULL auto_increment,  
    `nome_agente` varchar(30) NOT NULL default '',  
    `descr_agente` varchar(50) NOT NULL default '',  
    `location` int(11) NOT NULL default '0',  
    `current_terminal` varchar(100) NOT NULL default '',  
    `vnc_url` varchar(100) NOT NULL default '',  
    `sys_dt_creazione` datetime NOT NULL default '0000-00-00 00:00:00',  
    `sys_user_creazione` int(11) NOT NULL default '0',  
    `sys_dt_modifica` datetime NOT NULL default '0000-00-00 00:00:00',  
    `sys_user_modifica` int(11) NOT NULL default '0',  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```
• **Known Uses**: The “Updated When” pattern was found in all the following applications:
  
  00010.sql, 00029.sql, 00032.sql, 00038.sql, 00045.sql, 00049.sql, 00050.sql, 00059.sql,
  00068.sql, 00073.sql, 00085.sql, 00093.sql, 00244.sql.
  
  • **Related Patterns**: “Updated By”.

### 4.17 - The “User Group Association” Pattern

This pattern allows the definition of user association, usually to ease the management of rights and permission. 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 as needed.

• **Pattern Name and Classification**: User Group Association

• **Intent**: Detect which users have with similar rights.

• **Also Known As**: User Grouping.

• **Motivation (Forces)**: The most common use of this pattern, while not stored in a relational database, is the UNIX permission scheme. There are users, groups, then special commands to assign users to groups. When a data centric application wants to mimic this functionality and store it (naturally) in a database, this pattern provides the solution.
• **Applicability:** Any application that has support for user identification and group creations.

• **Structure:** The ER diagram in Figure 17 illustrates the pattern:

```
User ID

User Name

Group ID

User ID

Group Name

Figure 17. “User Group Association” in UML. This figure illustrates the pattern.
```

• **Participants:** The authentication module within an application will validate the user's credential against this users table, then, if valid, will use the user-group table to detect the group name, in order to present the appropriate functionalities.

• **Collaboration:** Other tables could just store the group name to model permissions and access rights.

• **Consequences:** A user interface will have to be developed to allow administrative users to populate this table.

• **Implementation:** Applications in need of complex access right management to a potential large number of users.
Sample Code:

```sql
create table user (
    uid integer identity primary key,
    name varchar(50),
    firstname varchar(50),
    uname varchar(20),
    unique (uname)
);
create table groups (
    gid integer identity primary key,
    gname varchar(20),
    unique (gname)
);
create table user_groups (
    uid integer,
    gid integer,
    primary key (uid,gid),
    foreign key (uid) references user(uid),
    foreign key (gid) references groups(gid)
);
```
• **Known Uses:** The “User Group Association” pattern was found in all the following
4.18 - The “User Preferences” Pattern

It is often required to allow a user to customize an application to his taste and needs. The “User Preferences” pattern will help in implementing this.

- **Pattern Name and Classification:** User Preferences

- **Intent:** Store user preferences.

- **Also Known As:** User Customization.

- **Motivation (Forces):** A sample application for this pattern is a web site that want to allow each logged in user to set his preferred color theme, or widget order. Each settings will given a name and a value. Then a table will store the user id, the settings name and it's value. Note that is not uncommon to store all values a string to simplify the table design.

- **Applicability:** Any application that can identify the users and support customization.

- **Structure:** The ER diagram in Figure 18 illustrates the pattern:
**Participants:** The same as the “System Preferences” DDP, but the content is related to user settings.

**Collaboration:** The “User Preferences” table is linked to the users table.

**Consequences:** This feature should be used with care, because developers will tend to use it excessively in order to prepare for eventual change in requirements.

**Implementation:** Any application with support for user authentication and requires to provide user-level parametrization.

**Sample Code:**

```sql
CREATE TABLE horde_prefs (  
pref_uid VARCHAR(255) NOT NULL,  
pref_scope VARCHAR(16) NOT NULL DEFAULT '',  
pref_name VARCHAR(32) NOT NULL,  
pref_value TEXT,  
PRIMARY KEY (pref_uid, pref_scope, pref_name)
);
```

```sql
CREATE MEMORY TABLE USERPREFS (  
NAME VARCHAR(16) NOT NULL PRIMARY KEY,
```
• **Known Uses:** The “User Preferences” pattern was found in all the following applications:
  
  00019.sql, 00020.sql, 00074.sql, 00281.sql.

• **Related Patterns:** None.

### 4.19 - The “User And Password” Pattern

Many application require a user name and a password to identify and authorize the user. In it's simplest implementation, a table host a user and the hash of the password.

• **Pattern Name and Classification:** User and Password

• **Intent:** Store the user credentials, the username name and the password in a database.

• **Also Known As:** The “User” can also be referenced as a “username”, “Login” and “sign on”. Some systems use an email address as username to guarantee unicity.

• **Motivation (Forces):** A database is a safe place to store the user credentials specially when the number of users grows.

• **Applicability:** For a data centric application where a database engine is already an essential part of the application.

• **Structure:** The ER diagram in Figure 19 illustrates the pattern:
• **Participants:** One table in the database having one characters column for the user and another for the password. The user column in unique and never null.

• **Collaboration:** The password entered is transformed to a hash, and query is run against the table with the given user and hash. If a record is found, the user is allowed to log in. This will prevent a system administrator with access to the data to see the passwords entered by the users. It is worth mentioning here that an anti-pattern was discovered, one that store passwords in clear. This is very dangerous and should never be done.

• **Consequences:** Special care should be taken when storing the credentials in a database as any user with read only permission can have access to this database. If the passwords are not hashed, and if the security settings for this table is set in a way to prevent non administrative users from looking into the database, there could be a huge security gap.

• **Implementation:** Both application number 55 and 232 and most of the applications where the pattern was found, also use the table to store additional information such as the email address and time stamps about the last login, the full name of the user, etc.

• **Sample Code:** By analyzing some the data retrieved from the online repositories, the pattern was found in huge number of scripts : The tables below shows an extract of the script 55 and 232, which were randomly picked for illustration, with the username and

---

**Figure 19.** “User and Password” in UML. This figure illustrates the pattern.
password field in bold. Below are a couple of examples showing how the pattern is implemented:

```sql
CREATE TABLE mantis_user_table (
    id int(7) unsigned NOT NULL auto_increment,
    username varchar(32) NOT NULL default '',
    email varchar(64) NOT NULL default '',
    password varchar(32) NOT NULL default '',
    date_created datetime NOT NULL default '1970-01-01 00:00:01',
    last_visit datetime NOT NULL default '1970-01-01 00:00:01',
    enabled int(1) NOT NULL default '1',
    protected int(1) NOT NULL default '0',
    access_level int(2) NOT NULL default '10',
    login_count int(11) NOT NULL default '0',
    cookie_string varchar(64) NOT NULL default '',
    PRIMARY KEY (id),
    UNIQUE KEY username (username),
    UNIQUE KEY cookie_string (cookie_string)
) TYPE=MyISAM;
```

```sql
CREATE TABLE [dbo].[Users] (  
    LogonID nvarchar(20) NOT NULL default '0',  
    Name nvarchar(40) default NULL,  
    Password nvarchar(20) default NULL,  
    EmailAddress nvarchar(40) default NULL,  
    LastLogon datetime default NULL,  
    PRIMARY KEY (LogonID)
)
```
• **Known Uses:** The user and password pattern was found in all the following applications:

00010.sql, 00017.sql, 00040.sql, 00049.sql, 00051.sql, 00055.sql, 00060.sql, 00065.sql, 00076.sql, 00079.sql, 00093.sql, 00164.sql, 00178.sql, 00181.sql, 00187.sql, 00191.sql, 00195.sql, 00203.sql, 00208.sql, 00209.sql, 00212.sql, 00215.sql, 00227.sql and 00232.sql

• **Related Patterns:** “User Group Association”.

### 4.20 - The “Workflow” Pattern

A Workflow is an application that allow user to fill a form and then move it thru 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 forms is stored in a single table. Each row is a data item, having attributes linking it to another table to define it's 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.

• **Pattern Name and Classification:** Workflow

• **Intent:** Allow the storing of attributes defined at run time.

• **Also Known As:** “Columns are Rows”

• **Motivation (Forces):** Workflows represent a very special type of application. The attributes types (text, type, number, decimal, dates, etc.) of any entity are defined at run-
time long after the database is designed. Also the number of attributes is often large (larger than the allowed number of columns), and variables (not all instances of the objects requires the same attributes). One solution to this problem is not to store the attributes as database columns as it is usually done, but as rows.

- **Applicability**: Any workflow or pivot table (Stathopoulou, & Vassiliadis, 2007).
- **Structure**: The ER diagram in Figure 20 illustrates the pattern:

  ![ER Diagram](image)

  *Figure 20. “Workflow” in UML. This figure illustrates the pattern.*

- **Participants**: Dynamic workflows, or when modeling pivot table.
- **Collaboration**: Other tables needs to group the data attributes in groups to refer to them.
  
  The primary key become the attributes type and the record instance id.
- **Consequences**: Tables grows in size very quickly.
- **Implementation**: All application that require to transform rows to columns and vice-versa.
- **Sample Code:

<table>
<thead>
<tr>
<th>N/A</th>
<th>00000.SQL</th>
</tr>
</thead>
</table>

  77
• **Known Uses:** The “Workflow” pattern was found in all the following applications:

• **Related Patterns:** None.

**4.21 - The “Standard Tables” Pattern**

The analysis of all the SQL scripts quickly showed that some typical tables are often present in many system implementations, below are some of those examples:

• MIME Types.

• Countries and Currencies, ISO codes, Time-zones, capitals, international phone extensions, etc.

• Languages with link to resource files to allow an application to run in many languages.

• Currency exchange rates (is offered as a commercial service by some financial institutions).

• **Pattern Name and Classification:** Standard Tables

• **Intent:** Store standard data.

• **Also Known As:** Common, Fixed Data Tables.

• **Motivation (Forces):** The most common example for this pattern is the storing of MIME (Multipurpose Internet Mail Extensions) types. Basically it help detecting which application should open which type of files depending on the file's extensions (the
sequence of characters after the last '.' in the file name.

• **Applicability:** Any data centric application can use these tables.

• **Structure:** The ER diagram in Figure 21 illustrates the pattern:

```
create table media_types (
    id    int auto_increment primary key,
    name  varchar(32) not null
);  
```

• **Participants:** The application deals with these tables as it does with any other.

• **Collaboration:** Dependent on each table and it's relationship with the rest of the database.

• **Consequences:** Known tables might require an integration effort to join the rest of the database.

• **Implementation:** All application modeling real-world entities.

• **Sample Code:**
CREATE TABLE IF NOT EXISTS mime (  
mimeid INT(9) NOT NULL AUTO_INCREMENT,  
type VARCHAR(255) NOT NULL DEFAULT '',  
description VARCHAR(255) NOT NULL DEFAULT '',  
PRIMARY KEY mime_pk (mimeid)  
) TYPE=InnoDB;

- **Known Uses:** The “Standard Tables” pattern was found in all the following applications:
  00007.sql, 00024.sql, 00088.sql, 00045.sql, 00058.sql, 00081.sql.

- **Related Patterns:** None.

**4.22 - The “Archived Data” Pattern**

If a table grows to a point where it hinder the performance of the application accessing it, it is sometimes split into a “master” and “archive”. Both would have the same structure, and the data is periodically moved from the master table to the archive, thus keeping it's size manageable. The application would then have to be modified in order to know when to query each.

- **Pattern Name and Classification:** Archived Data.

- **Intent:** Control the growth of data in large tables.

- **Also Known As:** Master File.
• **Motivation (Forces):** Imagine an point of sale application that has the sales history table growing out of control. To the point where it hinder the performance of the whole program. If the most accessed records are usually within the current month, it makes sense to keep only those records. To access historical data, and to prevent massive changes to the application, we want to keep the same design, the same columns names and types, so we create a clone table, with the “_ARCHIVE” extension added to it's name.

• **Applicability:** Any application with a few large tables, and access to the source of the application, since the application of this pattern still require modification to the applications using it.

• **Structure:** The ER diagram in Figure 22 illustrates the pattern:

![ER Diagram](image)

*Figure 22. “Archived Data” in UML. This figure illustrates the pattern.*

• **Participants:** There are no key duplicates. The data is simply moved from one table to another.

• **Collaboration:** All other table link to the master or archive table.

• **Consequences:** The application should look first in the master table, and if not found, look in archive table.
• **Implementation:** All application that needs to keep data grows under control, and where older data is not often accessed.

• **Sample Code:**

```sql
CREATE TABLE messu_archive (  
    msgId int(14) NOT NULL auto_increment,  
    user varchar(200) NOT NULL default '',  
    user_from varchar(200) NOT NULL default '',  
    user_to text,  
    user_cc text,  
    user_bcc text,  
    subject varchar(255) default NULL,  
    body text,  
    hash varchar(32) default NULL,  
    date int(14) default NULL,  
    isRead char(1) default NULL,  
    isReplied char(1) default NULL,  
    isFlagged char(1) default NULL,  
    priority int(2) default NULL,  
    PRIMARY KEY  (msgId)  
) TYPE=MyISAM AUTO_INCREMENT=1 ;
```

```sql
CREATE TABLE /*$wgDBprefix*/archive (  
    ar_namespace int NOT NULL default '0',  
    ar_title varchar(255) binary NOT NULL default '',  
    ar_text mediumblob NOT NULL default '',  
    ar_comment tinyblob NOT NULL default '',  
    ar_user int(5) unsigned NOT NULL default '0',  
    ar_user_text varchar(255) binary NOT NULL,  
```

82
• **Known Uses:** The “Archived Data” pattern was found in all the following applications: 00058.sql, 00073.sql.

• **Related Patterns:** None.

### 4.23 - The “Schema Software Version” Pattern

Add the version of the software to the schema name to allow many version of the same application to run on the same instance of the database.

• **Pattern Name and Classification:** Schema SoftwareVersion

• **Intent:** Allow the installation of many version of the same application on the same instance of the database.

• **Also Known As:** Variable Schema Name.

• **Motivation (Forces):** Imagine a developer, a tester or even a user who want to run many version of the same application on the same database instance. If this feature is not available, then a complete system installation and configuration is needed, without...
counting the user configuration changes needed to connect to the new version. This becomes more challenging if a rollback to an older version is to be performed. One simple solution to this problem is to add the version of the application in the database schema. All is needed when switching from one version of the application to another is to change the schema name in the application.

- **Applicability:** This pattern can be applied to any data-centric application.
- **Structure:** The ER diagram in Figure 23 illustrates the pattern:

![Figure 23. “Schema Software Version” in UML. This figure illustrates the pattern.](image)

- **Participants:** The application should accept the name of the schema in addition to the database connection string.
- **Collaboration:** None. All the table are in the same schema, and replicated in another.
- **Consequences:** Application should be built to run different versions on the same server. This could become quite challenging when different versions of a shared library are needed by different version of the software.
- **Implementation:** Applications that needs to allow more than one version of the software to run on the same instance of the database engine.
- **Sample Code:**

```sql
CREATE TABLE `#__templates_menu` (  `template` varchar(50) NOT NULL default '',
```
Each table in the script is preceded by the “`#__templates_menu`” prefix, which is replaced by the installer by the schema name.

Just like the 00212.sql, each table in the script is also preceded by a special string, “`/*$wgDBprefix*/`”, which is replaced by the installer by the schema name. Note that this application allow the user to define the schema.

- **Known Uses:** The “Schema Software Version” pattern was found in all the following
applications :
00212.sql,00073.sql.

• Related Patterns: “Software Version”.

4.24 - The “Record Locking” Pattern

If the application need to perform locking at the record level to prevent users from accessing some data items because they are being processed by another user, each record will contain a 'Locked' flag, a 'Locked By' id referencing a user, a 'Locked Time' and sometimes a 'Lock Limit' to automatically release a locked records after a predefined time.

• Pattern Name and Classification: Record Locking.

• Intent: Prevent access on the same data by other users or application.

• Also Known As: Logical Record Locking

• Motivation (Forces): Imagine a content management application where a user has to open an article and edit it. The editing might take some time, minutes, hours or even days. During that time, it should not be possible for other editors to edit the article. To prevent this, each article being edited will be marked as locked, with an owner (person who is editing the content).

• Applicability: Any application that needs to logically control concurrent access to any shared information.

• Structure: The ER diagram in Figure 24 illustrates the pattern :
Figure 24. “Record Locking” in UML. This figure illustrates the pattern.

- **Participants:** The application will allow access to the record only if it is not locked, or if the lock expired.
- **Collaboration:** This table is not related to any other.
- **Consequences:** This DDP should be modeled to the right tables.
- **Implementation:** Application in need to model data item locking, which, opposed to the built in locking mechanism to a database engine, might last as long as it needs.
- **Sample Code:**

```sql
CREATE TABLE `layouts` (  
  `id` int(11) NOT NULL auto_increment,  
  `name` varchar(100) default NULL,  
  `content` text,  
  `created_at` datetime default NULL,  
  `updated_at` datetime default NULL,  
  `created_by_id` int(11) default NULL,  
  `updated_by_id` int(11) default NULL,  
) 
```
CREATE TABLE tiki_hw_pages (  id int(14) NOT NULL auto_increment,  assignmentId int(14) NOT NULL default '0',  studentName varchar(200) NOT NULL default '',  data text,  description varchar(200) default NULL,  lastModif int(14) default NULL,  user varchar(200) default NULL,  comment varchar(200) default NULL,  version int(8) NOT NULL default '0',  ip varchar(15) default NULL,  flag char(1) default NULL,  points int(8) default NULL,  votes int(8) default NULL,  cache text,  wiki_cache int(10) default '0',  cache_timestamp int(14) default NULL,  page_size int(10) unsigned default '0',  lockUser varchar(200) default NULL,  lockExpires int(14) default '0',  PRIMARY KEY (studentName,assignmentId),  KEY id (id),  KEY assignmentId (assignmentId),  KEY studentName (studentName) ) TYPE=MyISAM;
**Known Uses:** The “Record Locking” pattern was found in all the following applications:

00010.sql, 00058.sql

**Related Patterns:** None.

**4.25 - Sample Implementation**

To illustrate how a database designer could use a great number of the DDPs covered above in a single application, a sample implementation is proposed for the modeling of a bookshop:
An experience database designer might see the design in figure 30 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
4.26 – Analysis of the Discovered Patterns

The sample implementation proposed in the previous section, and the nature of the functionality covered by each of the discovered patterns will help categorize them into six distinct sections:

System Settings – All the DDPs which 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.

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.

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” and “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 it's size grows beyond a certain threshold).

Semantic modeling - Knowing the attributes and relationships of the entities being modeled:

• The “Standard Tables” Pattern.

Workflow Proper Patterns – All DDPs implementing the required functionality for workflow functionality:

• The “Record Status” Pattern.

• The “Workflow” Pattern.
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 covered by this thesis (and the bookshop model presented earlier) 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 worlds application. 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 need to be put to build the relationship between the attributes in the model, define the constrains, 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 (like $K^{th}$ order statistics ) could be used to detect the upper and lower boundaries for each numerical attribute (bounded data). Finally, list detection will allow the detection of non bounded attributes (grouped data).

A new method for the definition of user requirements could be proposed to allow the business users to define their requirements in order extrapolate a database model. A database designer language, which would play the role of a reconnaissance agent in order to gather as
much information as possible on the model being built. That fifth generation language will be called DDRL for Database Designer Reconnaissance Language.
CHAPTER FIVE

CONCLUSIONS AND RECOMMENDATIONS

5.1 - Recommendations

The relatively high number of discovered design patterns in data-centric open source applications shows that the reviews database design encompass two aspects:

- Model Specific Attributes:
  All the tables, attributes, descriptions related to the entities being modeled.

- Common Practices:
  All the techniques, artifacts, recommendation for implementing the model.

It could be taught that design patterns could at most help in implementing the second point, as previously discovered design patterns in other disciplines such as object oriented programing by Gamma et al. (1995) and workflow by Dumais, et al. (2005) could not do better. But databases are different. The 'known lists' patterns which is a subset of the functionalities offered by semantic databases by Brown (1993), is a proof that those systems could be more 'aware' of the entities being modeled and the relationships they have with each other. The logs pattern is another example. With the recent availability of APIs dedicated to logging, it becomes easy for an application to asynchronously log all its messages to a dedicated table, which will make remote debugging easier.
Just like object oriented DPs, the presence of DDPs is a sign of a good application design. But this thesis cannot pretend that DDPs are the only needed constituent of data-centric application. The entities being modeled are not covered by any of the covered design pattern. Due to this fact, any design artifact will have to be categorized into two sections:

1. Model Related: All artifacts related all the attributes that are proper to the problem being modeled.
2. Feature Related: All artifacts that could be implemented by a DDP.

All the work done on DDPs was motivated by pedagogical concerns: To help people understand databases and make better designs. In this thesis, a new tool will be proposed, one who can interact with an end user thru a friendly interface, and which sole purpose is to design and fill a relational database by automatically implementing the DDPs while the model related features are generated from the user's input. The tool is an interpreter for a new 5th generation programming language called the Database Designer Reconnaissance Language, or DDRL.

5.2 - The Target Design Characteristics

In order to be appealing to the enterprise, DDRL should have the following minimum characteristics:

• Easy enough to be used by any person who knows how to use a word processor.

• Generate a relational design of the first normal form (but null values will be allowed) by Codd, (1984).

• Automatically detect the data domain of each attribute (integers, decimals, date/time and text).

• Manage the precision of each attribute to be as accurate as it needs to be.
• Detect the range for all numeric and date attributes.

• Automatically alter the database structure when the user's input patterns start to change.

• Automatically generate the constrains for each attribute, "learning" from the user's input.

5.3 - DDRL Stages

5.3.1 - Phase 1 User Identification and Authorization

The DDRL interpreter implementations should support user authentication, through the single sign-on feature of the JAVA Authentication and Authorization Service (JAAS), in order to associate a unique numerical ID to each user. That number will be known as the user id. This will allow the interpreter to associate a person to all database writes.

DDRL support two levels of users:

• The designer, allowed to enter data that might modify the data model.

• The filler, allowed only to enter data that passes the validation process.

5.3.2 - Phase 2 Parsing

The language is simple. The user start by entering a title (to become the table name), then a series of attributes in a [Description]:[Value] or [Description] > [Value] format. Each entry is on a separate line, and each line is terminated by two line feeds. The end of record is detected when the user enter an attribute description and value another time. The difference between a ':' separator and a '=>' is that the later hints the interpreter that the data that follows is part of list.
**Figure 25.** Flow Diagram of the DDRL Parser.

<table>
<thead>
<tr>
<th>Corporate Customers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First:</strong> Georges</td>
</tr>
<tr>
<td><strong>Age:</strong> 38</td>
</tr>
<tr>
<td><strong>Preferred Gadget &gt; iPad</strong></td>
</tr>
</tbody>
</table>

**Figure 26.** DDRL Input Example.

All the metadata generated from the user's input is stored in a dedicated schema, DDRL_MODEL. There are two main advantages for using a dedicated schema for storing metadata instead of relying on those provided by the database engine:

- Databases managed by DDRL can coexist with other applications under the same database instance. (Another factor to increase the acceptance).
- Statical data can be gathered and stored in this model in order to detect potential invalid data entries.
- The migration from DDRL will be easier to automate.
5.3.3 - Phase 3 Domain and Constraints Definition

In the second phase, the user's input is analyzed. For each attribute, on each line, the following checking are performed:

1. Is it an integer?
2. Is it a decimal?
3. Is it a form of any known date format? The JAVA libraries offer a wealth of date format. DDRL loop thru all the date format for all the locales, to try to find a match.
4. If all of the above checking fails, the free text analysis is invoked:
   1. Does the attribute value match a known real word entity (countries, currencies, ISSN, etc.)?
   2. Does it match any of the known regular expressions?
   3. Was the attribute previously assigned, in another record, the same value?
   4. Is it consistent in length, character and number distribution, compared to previous
values assigned to the same attribute?

In order to answer these question, the DDRL interpreter will prompt questions to the user. These questions will be referred to as 'Hints'.

5.3.4 - Phase 4 Hinting

The processing of the user's input might trigger a 'Hint', which is a question with a list of possible answers, generated by the parser, when a decision is needed before continuing. The user will answer the Hinter's feedback so that the parsing can continue.

Each attribute is given a hint level (HINT_LEVEL attribute in the TABLE_METADATA table in the DDRL_MODEL), which will set a confidence level on constrains and domain definition.

One of the most interesting patterns found during the analysis of open source databases, is the 'know lists' pattern. Different database applications often model similar real world entities, and therefore need similar attributes by Brown, (1993). Common 'know lists' are countries, salutations, units of measures, currencies, etc. Such lists could be provided by default before any software development starts, and could even be reused by many applications.

The concept could be taken a little bit further. Instead of storing values in these lists, it could be regular expressions. This will be add more control on the users's input.

If the user's input does not match against any know list or regex by Aho, (1990), it might be a member of an unknown list. To detect it, an SQL DISTINCT query is executed against previous entries for the same attributes. If previous records were assigned the same value, the scout will propose to transform the attribute to a list, create a new table, and replace user entered values with the key of the newly created table, thus establishing a many to one relationship.

Finally, if all of the above checking failed, DDRL will check if it matches the auto
guessed simple regex from previous entries.

5.3.5 - Phase 5 Precision management

One of the most challenging aspects of DDRL, is the management of the precision, and thus the records length and scale. As DDRL aims for a very neat design, it would be incorrect to have all text fields set to 255 characters, and all integers to 9 digits. A tight precision will also help in detecting incorrect data entries by hinting the user for any entry that overflow a certain attribute.

To precision of a field is defined and update from the user's input. If a text is entered the first time as 4 characters, the attribute will be defined as VARCHAR(4). If another input is entered as 10 characters, the attribute will be altered to become VARCHAR(10). The same goes for integers and decimals. The later benefits from special treatment, since the interpreter should keep track of the size of the integer (precision) and the decimal part (scale). The precision of each attribute is stored in the DDRL_MODEL schema, in the TABLE_METADATA table.

5.3.6 - Phase 6 Persisting the Data

5.3.6.1 - Persistence Process Details

If the table does not exist, it will created during the persistence operation. Each table created by the DDRL will always have the following extra columns:

• ID : Primary key, automatically incremented by the database engine for each new record.
• DATE_CREATED : The date and time when this record was created/inserted into the table.
• CREATED_BY : An integer representing the ID of the user that inserted that record.
• DATE_UPDATED : A time stamp of when the record was last modified.
• UPDATED_BY: An integer representing the ID of the user that last updated that record.

5.3.6.2 - Persistence Logging

All persistence operations will be noted in the logging tables. These tables simply store who changed what value to what.

Persistence Logging

The logging tables are stored in the DDRL_MODEL schema and structured as follows:

• SCHEMA_NAME: The name of the schema of the modified field.
• TABLE_NAME: The name of the table of the modified field.
• COLUMN_NAME: The name of the modified field.
• COLUMN_TYPE: The SQL data type of the modified field.
• VALUE_BEFORE_CHANGE: The value, stored as a string, of the field value, before it was changed.
• VALUE_AFTER_CHANGE: The value, stored as a string, of the new field value.
• CHANGED_TIMESTAMP: The date and time of the change.
• USER_ID: The id of the user who made the change.
• QUERY_ID: A integer used to uniquely identify the query that performed the logged change.

Many database engines are provided with their own implementation of database logging, but these are often difficult to access are proprietary for each vendor. Implementing the loggers at the interpreter level allow the later to make real time elaborated statistics on the data evolution.

5.3.7 - Phase 7 Fields Prompting

Once all the previous phases are executed, when the user enters an items name, DDRL
will look in the model for all tables that have the same description as the one entered, and prompt the user for selection. Once done, the interpreter will generate a classic data entry form with the possibility to add fields.

5.4 - Asynchronous Data Analysis Modules

The server side components of DDRL is a scheduler that will periodically and independently of any user interaction, execute a set of modules who's main objective is to analyze the content of the database. These modules should be implemented as plug-ins, to allow independent entities to develop new functionality or override the default ones. The following basic modules are required for correct operations:

5.4.1 - Data Limit Detection Module

Although the definition of data types like integer, text, dates, etc, will prevent a user from entering text where a number is expected, it cannot, for example, prevent a user from defining the age of a person as 999 years, if the age was defined as a 3 digits attribute. The Data Limit Detection Module (DLDM) periodically analyzes all the numerical data and using the kth order statistics by David, et al. (2003) and by Sefling, (1980) will detect the k minimum and maximum values for comparable fields. When the user enters a value within or beyond the maximum and minimum ranges, a warning hint will be prompted for the user. If approved, the next scheduled run of the DLDM will update the kth limits. The "business logic" would have been updated without involving a change management process.

5.4.2 - Permission Detection Module

This module will periodically analyze the history of changes to the databases in order to propose a security map. The idea behind this module is the following:
Since DDRL does not initially enforce a security model, all the users accessing the database will be able to change any record. This will force the first users to enforce a discipline on who will change which data elements. This module is designed to detect such behavior and propose it for enforcement. This is done by applying the following query on the logging table:

```
Select the distinct and count of update combinations (group of columns changed in each modification query) for each user. Users performing the same changes will be grouped into the same "permission groups".
```

### 5.4.3 - Workflow Detection Module

This module will try to detect if the users of interpreter are trying to model a workflow. It will analyze the output of permission module, and will order the generated permission groups by the timestamp stored in the logging tables. If a chronological sequence of permission groups is repeatedly detected, it will be proposed as a workflow task.

### 5.4.4 - One to One Relation Detection Module

This module will try to locate a null group of attributes or two permission groups for the same table, in order to split it and establish a one to one relationship. The primary key will be replicated in both tables.

### 5.5 - Update and Change Management

Once the first table is created, the DDRL interpreter should allow the user to add more records without having to enter the full description of each attribute. So for a table called 'persons', with the attributes first, middle and last, when the user types 'persons' and press enter, the interpreter will display all the attributes and move the cursor to the first one:
The user will fill in values for any of the attribute values. The user may skip some, or even add a new attribute:

- First: Georges
- Middle: Pierre
- Date of Birth: 3 June 1973

As described earlier, the persistence function will then alter the 'persons' table and add a new 'Date of Birth' column. The hinting service would have detected and checked that the field is always a date, and the column will be created as an SQL date.

If the data format of an attribute is modified by a 'designer' user (Section 3.1), by an entering say a text where an integer is expected, the hinter will warn the user of the inconsistency. If the user confirms it, the interpreter will modify the table from integer to “varchar”, and will transform all integers to string.
5.6 - J-DDRL

J-DDRL is an open source (BSD license) prototype implementation of a DDRL interpreter in Java with a MySQL backend. The choice of Java for implementing DDRL is obvious. Till this day, it is the only real platform independent language, which makes it ideal for operating in any environment. Java also benefits from very rich class library that will make the implementation much easier than other languages.

But the main reason for choosing the JAVA language, is to increase the acceptance and therefore the popularity of DDRL by allowing it to work with any database engine, thanks to JDBC, in any environment., thanks to its unmatched portability. It will make it less of an “alien” product, if it could effortlessly integrate with the existing IT infrastructure.

As previously covered, DDRL store the database metadata and constrains in the tables of the DDRL_MODELL schema. This will make the migration from DDRL to a fully fledged data-centric application much easier.

The J-DDRL implementation will be executed in four layers:

1. UI Layer: Handle user interaction and control the user's input and the corresponding DDRL output.
2. Hinting: Part of the UI, to handle the feedback from the persistence layer.
3. Parsing Layer: Parse the user's input.
4. Persistence Layer: Save or load the user's input to the database tables.

The parsing and persistence layers can be distributed separately from other modules, to allow developers to use is as a 'Facade' to the JDBC API. This will allow DDRL to be easily integrated into other applications. The tricky part would be the implementation of the Hinting interface. But since the GUI or Web framework are finite and well know, it would be possible to implement
Hints in Swing, SWT, console, web, etc.

5.7 - CONCLUSION

The mechanism on top of which DDRL is built is a new approach where the end result, which is often too complex to be expressed, is achieved by progressively asking a non technical user, the right questions at the right time.

The data models produced by DDRL are not intended to replace fully fledged data-centric applications, yet they will provide valuable feedback and starting point for creating one, all at a minimal effort. A power user should be able to start using it in less than an hour. The people using it will benefit from a partial automation, which is often enough to increase productivity.

5.8 - Further Work

The modules covered in this thesis are only a few of what could be achieved with DDRL. Additional work could be done to add support to the definition of business logic thru an english-like programming language. It would be interesting to perform some data mining on the related records to try to detect calculated fields.
References


