J-DDRL: Java Data Definition Reconnaissance Language

Ramzi A. Haraty, Georges Stephan
Lebanese American University
Department of Computer Science and Math
Beirut, Lebanon
Email: {rharaty, geroges.stephan}@lau.edu.lb

Hussein Mohsen
Indiana University
School of Informatics and Computing
Bloomington, IN, USA
Email: hmohsen@indiana.edu

Abstract

Software development is a long and expensive process; so much that organizations often end up with partial computerization. This paper proposes a new high level design language (a kind of fifth generation language), DDRL, for Database Designer Reconnaissance Language, which would allow any person who know 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 portability and database connectivity features of JAVA make it ideal, if not unique platform for implementing this concept. J-DDRL is a reference, open source (GPL-3), implementation of DDRL, which could be a starting point for a community effort to further evolve the concept.

keywords: Databases, Design Patterns, and Reconnaissance Language.

1. Introduction

All software development processes, and especially for data centric applications, starts with the gathering and analysis of the user's requirements. Many techniques such as the software requirement specification (SRS) document [1-4] were proposed, but all approaches are time and resource consuming; and thus, expensive. Software developers will not initiate this process unless the project is officially started.

The time wasted thru the long and iterative process of the gathering of business requirements could be better used. Since the expansion of open source software, and after studying the majority of the large open source application database designs, numerous patterns emerged [5-6]. 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 paper will introduce a new programming language, DDRL (Database Designer Reconnaissance Language), which is used to define these "subtleties". What makes DDRL unique compared to other languages such as SQL, is that it is intended to be used by an end user, not a programmer.

The relatively high number of discovered design patterns in data-centric open source applications shows that the reviewed database designs encompass two aspects [7]:

- 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 thought of 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. [8] and workflow by Dumas, et al. [9] 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 [10], 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 [11].

Just like object oriented DPs, the presence of DDPs is a sign of a good application design. But this work 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:

- Model Related: All artifacts related all the attributes that are proper to the problem being modeled.
- 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 [12-13]. In this work, we propose a new tool - 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 fifth generation programming language called the Database Designer Reconnaissance Language, or DDRL. We use JAVA [14] to implement DDRL.

The remainder of the paper is organized as follows: section 2 presents the design characteristics and stages of DDRL. Section 3 offers the data analysis modules and update and change management. Section 4 highlights J-DDRL and section 5 presents the conclusion.

2. DDRL Stages

DDRL has the following 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 [15] (but null values will be allowed).
- 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 constraints for each attribute, "learning" from the user's input.

The different stages of the DDRL are as follows:

Phase 1: User Identification and Authorization

The DDRL interpreter should support a form of user authentication, in order to associate each user with an integer. That number will be known as a user id. This will allow the interpreter to associate a person to each database operation [16]. DDRL supports two kinds of users:

- The designer - allowed entering data that may modify the data model.
- The filler - allowed only entering data that passes the validation process.

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. The end of record is detected when the user enters a different attribute description and value. The difference between a ':' separator and a '>' is that the later hints the interpreter that the data that follows is part of list. The parser functionality is depicted in Figure 1.

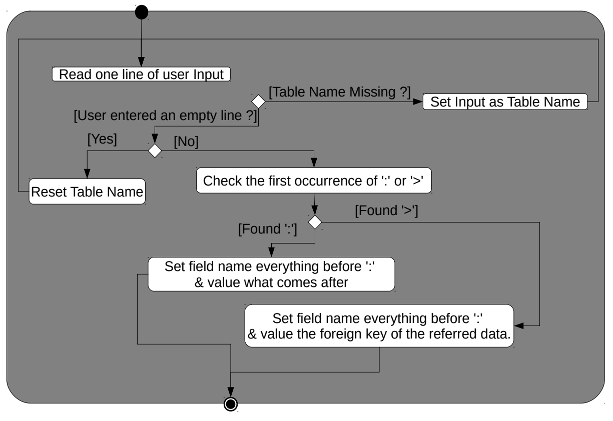
Read one line of user Input
[Table Name Missing ?]
Set Input as Table Name
[User entered an empty line ?]
[Yes]
[No]
Reset Table Name
Check the first occurrence of ':' or '>'
[Found '>']
[Found ':']
Set field name everything before ':' & value what comes after
Set field name everything before ':' & value the foreign key of the referred data.

Figure 1 - Flow diagram of the DDRL parser

The first phase is the parsing of the user's input. The user input could look like this:

Corporate Customers

First:Georges

Age : 38

Preferred Gadget > iPad

Phase 3: Domain and Constrains Definition

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

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

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

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. If the hinter detects that the domain and constrains of an attribute are well-defined, the hinter will flag the attribute so that it will not ask further the user about it.

All the metadata generated from the user's input is stored in a dedicated schema, DDRL_MODEL, which is depicted in Figure 2.

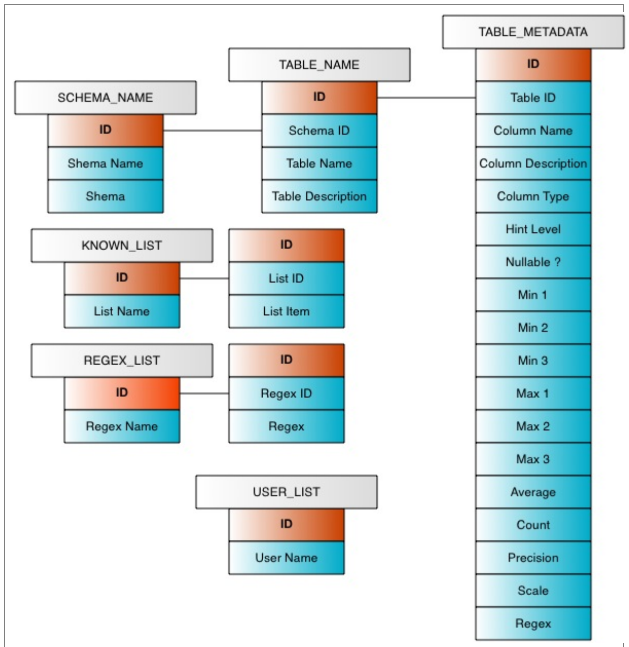

Figure 2 - The main tables in the DDRL_MODEL schema

The DDRL_MODEL stores the Meta data for all the schemas managed by DDRL. There are three 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.
- Statistical data can be gathered and stored in this mod-el in order to detect potential invalid data entries.
- The migration from DDRL will be easier to automate.

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. 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 bit further. Instead of storing values in these lists, it could be regular expressions. This will be adding more control on the user's input. If the user's input does not match against any know list or regex, 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.

Phase 5: Precision Management

One of the most challenging aspects of this concept 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 nine digits. A tight precision will also help in detecting incorrect data entries by hinting the user for any entry that overflows a certain attribute. If the user approves a large entry, the precision will be updated and the database table modified accordingly.

Phase 6: Persisting the Data

Persistence Details

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

Persistence Logging

All persistence operations will be noted in the logging tables. These tables simply store who changed what value and to what. 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 and are proprietary for each vendor. Implementing the loggers at the interpreter level allows the later to make real time elaborated statistics on the evovled data.

Phase 7: Fields Prompting

Once all the previous phases are executed, and 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. It then prompts the user for selection. Once done, the interpreter will generate a classic data entry form with the possibility to add fields.

3. 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 whose main objective is to analyze the content of the database. These modules are 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:

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 k^{th} order statistics [14], 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 k^{th} limits. The "business logic" would have been updated without involving a change management process.

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 in the same "permission groups".

Workflow Detection Module

This module will try to detect if the users of the 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, if will be proposed as a workflow task.

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.

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 presses enter, the interpreter will display all the attributes and move the cursor to the first one:

Persons[Enter]

First:

Middle:

Last :

The user will fill in values for any of the attribute values. The user may skip some, or even add a new attribute:

Persons

First: Georges

Middle : Pierre

Date of Birth : 3 June 1973|

Last :

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, by 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.

4. J-DDRL

J-DDRL is an open source reference implementation of a DDRL interpreter in Java with a MySQL backend [17]. 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.

As previously covered, DDRL stores the database metadata and constrains in the tables of the DDRL_MODEL 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:

- UI Layer: Handles the user interaction and controls the user's input and the corresponding DDRL output.
- Hinting: Part of the UI, to handle the feedback from the persistence layer.
- Parsing Layer: Parses the user's input.
- Persistence Layer: Saves or loads the user's input into 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-known, it would be possible to implement hinters in Swing, SWT, console, web, etc.

5. 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 ones, all at a minimal effort. A power user should be able to start using it in short period of time. The people using it will benefit from a partial automation, which is often enough to increase productivity.

The modules covered in this work 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

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

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

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

[4] Vasutiu, V., Vasutiu, F. *Database Design Patterns*. Ph.D. Dissertation. University Szeged, Szeged, Hungary, 2006.

[5] Vitacolonna, N. Conceptual Design Patterns for Relational Databases, *Proceedings of the 17th International Conference on Information and*

Software Technologies, Kaunas, Lithuania. pp. 239 – 246, 2011.

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

[7] Google Code Search. Retrieved on September 28, 2012 from http://code.google.com/apis/codesearch/docs/2.0/reference.html, 2012.

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

[9] Dumas, M., Van der Aalst, W.M., & ter Hofstede, A. *Process Aware Information Systems: Bridging People and Software through Process Technology*, John Wiley and Sons, Hoboken, New Jersey, USA, 2005.

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

[11] Haraty, R. A. and Stephan, G. (2013). Relational Database Design Patterns. *Proceedings of the IEEE Second International Conference on Big Data Science and Engineering (BDSE 2013)*. Sydney, Australia. December 2013.

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

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

[14] Java Authentication and Authorization Service (JAAS) Reference Guide (2012). Retrieved on September 28, 2012 from http://docs.oracle.com/javase/1.4.2/docs/guide/security/jaas/JAASRefGuide.html, 2012.

[15] Codd, E.F. Is your DBMS Really Relational? *ComputerWorld*, (10), 1984.

[16] Brajendra Panda, William Perrizo and Ramzi A. Haraty. Secure Transaction Management and Query Processing in Multilevel Secure Database Systems. *Proceedings of the Symposium on Applied Computing*. Phoenix, AZ, pp. 363-368, April 1994.

[17] MySQL Open Source Database. Retrieved on September 28, 2013 from http://www.mysql.com/, 2013.

Biographies:

Ramzi A. Haraty is an associate professor of Computer Science in the Department of Computer Science and Mathematics at the Lebanese American University in Beirut, Lebanon. He is also the academic and internship coordinator for Middle East Program Initiative's Tomorrow Leader's program. He received his B.S. and M.S. degrees in Computer Science from Minnesota State University - Mankato, Minnesota, and his Ph.D. in Computer Science from North Dakota State University - Fargo, North Dakota. His research interests include database management systems, artificial intelligence, and multilevel secure systems engineering. He has well over 110 books, book chapters, and journal and conference paper publications. He supervised over 110 dissertations, theses and capstone projects. He is a member of the Association of Computing Machinery, Institute of Electronics, Information and Communication Engineers, and the International Society for Computers and Their Applications.

Georges Stephan received his Masters of Science in Computer Science from the Lebanese American University – Beirut, Lebanon. His research interests include database systems.

Hussein Mohsen holds a B.S. in Computer Science from the Lebanese American University (LAU) in Beirut, Lebanon, 2011. He was an Erasmus Mundus graduate exchange student in Bioinformatics at Newcastle University in Newcastle upon Tyne, UK in 2012-13, and he is now pursuing his M.S. in Computer Science/Bioinformatics at Indiana University in Bloomington, Indiana, USA as per the Fulbright program. He is a co-author of a book and journal publication about speech recognition and speech synthesis, and his research interests include machine learning, bioinformatics and transaction processing. Hussein is a member of the Association for Computer Machinery (ACM).