On Pattern-Based Database Design and Implementation

Erki Eessaar
Department of Informatics, Tallinn University of Technology, Raja 15, 12618 Tallinn, Estonia
essear@staff.ttu.ee

Abstract

Patterns describe common solutions to recurring problems. This paper presents a pattern-based approach of improving database design and automating generation of statements that are written in a database language. During the logical design of a database, a database designer has to find domain-independent database design patterns that are applicable in the database. The selected patterns are used in order to generate code during implementation of the database. For example, this approach can be used in order to generate implementations of integrity constraints and updatable views. We present a pattern-based and template-based code generator that is created specifically for the development of databases.

1. Introduction

An information system (IS) is the main functioning and learning environment of a subject of information society. Development of an IS is subsequently the central role of the subject [1]. Therefore, a modern sustainable IS must have built-in support for its continuous evolution. It would be helpful if we could automate development of an IS as much as possible in order to make implementation process easier and faster, reduce errors in the code, enforce coding standards, and reuse best practices.

The idea of automating software design is not new [2]. The OMG proposal of Model Driven Architecture (MDA) [3], which allows developers to implement systems by using model transformations, has yet again concentrated attention to this field. However, Olive [4] concludes that the goal of automating software design "has not been achieved in a satisfactory degree."

In this paper, we are interested in the automation of database code generation. We propose an approach that focuses on the development of data-intensive systems. The use of this approach should simplify system development and improve data quality in a database. The approach promotes the use of database languages.

The concept "data model" is semantically overloaded. Among other things, this concept denotes abstract, self-contained, logical-definition of an abstract programming language [5]. A database designer can use this language in order to design many different databases. In this paper, we consider the underlying data model of SQL:2003 [6]. This data model is one possible approach of object-relational (OR) data model and we call it "ORSQL". We call a database system (DBMS), the underlying data model of which is ORSQL, as ORDBMSQL.

A pattern describes a family of solutions to a recurring problem [7]. The first goal of the paper is to propose a pattern-based database design and implementation approach that promotes the use of patterns as the basis of code generation. Therefore, the second goal of the paper is to present a software system that allows us to perform code generation based on database design patterns.

The rest of the paper is organized as follows. Firstly, we investigate the state of the art of ORSQL database design and implementation in order to explain the motivation of this study. Secondly, we propose a pattern-based database development approach. Thirdly, we present a pattern-based and template-based code generator that can be used in order to generate statements in different database languages. These statements implement database design patterns. Finally, we draw conclusions and describe the future work.

2. The state of the art of automation of ORSQL database development

Many CASE environments allow us to create pictorial representations of the design of a database by using UML or other languages. Some of these environments allow us to generate database language statements based on a design specification. This is a kind of model driven development. Usually the resulting code consists of statements for creating static
designing a database of products presents 13 Entity Relationship diagrams about different types of constraints. For instance, Silverston existing conceptual data models use only relatively few specification. Date [5] notes that database users must be able to always trust the results of database queries and therefore writes about integrity constraints: "they're what database systems are all about". Data quality problems affect an organization at the operational, tactical, and strategic level according to Redman [10]. Unfortunately, the study [11] shows that existing databases do not follow database design guidelines and best practices very well. For example, Blaha [11] studies 35 databases. He notes that about 75% of developers enforced primary key constraints and 10% databases declare foreign key constraints. What could be the reasons of such practice?

Constraints that are enforced in a database are actually business rules [5] and they must be identified during the creation of a conceptual data model [12]. UML has limitations in specifying conceptual constraints explicitly compared to some other modeling languages. For instance, Object Role Modeling allows designers to use 13 types of constraints in a data model [13]. Seven of them have an incomplete corresponding construct in UML and three of them do not have a corresponding construct in UML at all [13]. It is possible to specify constraints in UML models by using Object Constraint Language (OCL) [14] or other languages. There are studies about how to specify integrity constraints in UML conceptual data models by using OCL [12] and about generation of database language statements based on OCL constraints [15]. However, many current software tools do not support this kind of code generation [8]. In addition, some studies have pointed to the complexity and limitations of OCL. For instance, Mandel and Cengarle [16] come to conclusion that OCL is not as expressive as relational calculus and cannot compute all the recursive functions. In addition, Date [17] demonstrates the imprecision of some parts of the OCL specification.

The result of this state of affairs is that some existing conceptual data models use only relatively few different types of constraints. For instance, Silverston [18] presents 13 Entity Relationship diagrams about designing a database of products. In case of associations he specifies participation and cardinality (always one or infinity) at the association ends and exclusive arc constraints. In case of generalizations, he specifies completeness constraint: "the sum of subtypes covers the supertype in its entirety" [18]. In addition, he requires disjointness within each set of subtypes and permits non-mutually exclusive sets of subtypes. For example, Miliauskaite and Nemuraite [12] present nine different types of constraints on relationships but the models of products of Silverston [18] use only three of them. Nevertheless, each system that is built up based on the models of products should enforce these rules.

Updatable views allow us to implement encapsulation of a database. Views provide the public interface that can be used by programs in order to read and modify data. There exist systematic approaches of view updating [19]. On the other hand, the SQL standard [6] specifies a lot of restrictions to the updatable views and existing ORDBMS SQLs have even more restrictions [20]. Some ORDBMS SQLs allow designers to create trigger procedures or query rewrite rules (rules) in order to implement updatability of views [20].

Existing ORDBMS SQLs have limitations in the context of creating declarative integrity constraints [19, 20, 21]. Often a database designer has to use an imperative language in order to develop a set of trigger procedures that are executed due to some event [20, 21]. It is also possible to implement constraints in the stored procedures that are recorded in a database or in the applications. However, in this case it is possible to violate the constraints if we do not use a particular procedure or application. Database systems that provide snapshot isolation for concurrency control are prone to constraint violation that is a generic and important type of concurrency anomaly [22]. Therefore, trigger procedures that implement constraint checking may have to implement serialization [21]. It adds another layer of complexity to the code.

In conclusion, current widespread design languages and tools do not provide the best possible support to specification of integrity constraints and updatable views. Current ORDBMS SQLs enforce development of complex imperative code. A result could be that programmers avoid the use of features of a DBMS and instead try to implement all the necessary functionality in applications. We need methods and tools in order to improve database design and encourage the use of all the features of a DBMS.

3. Pattern-based database design

The evolution of an IS can be based on agreed patterns that describe accepted solutions to recurring problems. Each pattern addresses "one or a common set of design problems in a certain context" [23]. We propose to use domain-independent database design patterns. Based on a definition of domain [24] it means that these patterns can be used for designing databases for different spheres of knowledge, influence, or activity (store, IT company, university etc.).
Each pattern is associated with a set of strategies [23]. Each strategy specifies how to implement the solution to the design problems by using a particular software system (platform) [23]. In case of the database design patterns the platform is a version of a database system (DBMS).

We define the strategy of a database design pattern as a set of sets \( \{E, T, P\} \). \( E \) is a set of templates that can be used in order to construct statements that explain the strategy. These statements can be in a natural language or could use some formal notation like [21]. \( T \) is a set of templates that can be used in order to construct statements in a database language that is available in a particular version of a DBMS. Conceptually, we can think about a template as a truth-valued function that has a set of parameters. \( P \) is the set of all the parameters that are used by templates in \( E \) and \( T \). A user of a strategy has to give values to its parameters. These values must be used in order to construct statements based on templates in \( E \) and \( T \). This construction can be performed by a software system (see section 4). Each constructed statement of a database language evaluates to TRUE only if it can be successfully executed in a database.

Related patterns can form a pattern language. The use of patterns allows organizations and individuals to concentrate on the accumulation of new knowledge instead of searching solutions of the same problems over and over again. The use of patterns allows developers to accelerate IS development process and improve the quality of the products of this process. Patterns and pattern-based generators of software systems allow developers to evaluate the results of system development and improve them if necessary.

Existing studies about patterns pay a lot of attention to the design of applications. For instance, there exist object-oriented design patterns that form GRASP (General Responsibility Assignment Software Patterns) language [25] and systems that generate code based on object-oriented design patterns [7, 26].

On the other hand, few studies consider database design patterns. There exist catalogues of reusable and mostly domain-specific specifications of databases (like [18, 27]). However, these specifications are not presented as patterns. We can also use analysis patterns (like the ones presented in [28]) as a basis in order to construct conceptual and logical data models. There exist pattern languages (like Stars and A Pattern Language for Smalltalk & Relational Databases) [29] that specify some aspects of relational database design. However, these small languages do not take into account different OR data model approaches and do not consider all the aspects of database design (for instance, when to choose natural or surrogate key).

### 3.1. A pragmatic database design approach

Next, we propose a pragmatic database design approach (see Figure 1). It can be used if an organization wants to use model driven development and has acquired a CASE environment that provides limited support to the generation of code based on a database design specification. It makes possible to continue the use of existing systems and get the benefits of code generation. This approach can be used in the context of different software development methods. For instance, the process could be executed repeatedly during an iterative development.

The approach is an extension of the well-known database design methodology [30] according to which database design consists of conceptual design, logical design, and physical design. We propose that during the logical database design, database designers have to identify database design patterns that should be used in the particular database. In case of each occurrence of the use of a pattern, a designer has to document pattern name, involved data structures (names of tables and columns in case of ORSQL), and comments. Designers must use information about constraints, which is collected during conceptual database design, as a basis of the pattern selection.

![Figure 1. The proposed approach](image)

We assume that during implementation of a database it is possible to generate some code based on a physical data model by using a CASE environment. Additional code must be generated based on the identified patterns by using a template-based and pattern-based code generator (see section 4). We think that the proposal to use a set of relatively simple environments instead of one sophisticated environment is in line with the current trend of using agile system development methods and in particular Agile Model Driven Development (AMDD). AMDD advocates the use of simple, flexible tools if they help to achieve the goals of a developer [31].

Model Driven Architecture (MDA) proposes the sequential and transformation-based creation of the computation independent model (CIM), the platform
corresponding patterns. Next, we present an

Therefore, each constraint type has zero or more

involved in a constraint in more than one way.

implementation.

It allows us to facilitate their
different projects. It enables us to implement and therefore it is possible to continue the use of patterns in case of a particular DBMS.

On the other hand, strategies are "PSM-specific". Each strategy of a database design pattern specifies how to implement the pattern in a database that is created by using a particular DBMS. Specifications of strategies are used in order to generate code.

MDA relies on the use of UML. Our proposed approach does not require the use of UML during database design (although it is not prohibited). It is also not necessary to use UML in order to specify a pattern (see section 3.2). Each strategy is specified in terms of statement templates and does not use UML.

One important difference between the pattern-oriented analysis and design process (POAD) [26] and our database design approach is that they concentrate on analysis/design and design/implementation phases, respectively. POAD proposes to use patterns as building blocks that are selected during analysis. The selected patterns are decomposed and refined during design phase. On the other hand, the approach that is presented in this paper proposes to use patterns in order to annotate database design models. The approach does not prescribe how these models must be created and therefore it is possible to continue the use of existing modeling methods and tools.

In section 2, we stressed the importance of integrity constraints and updatable views. We have created patterns for different types of integrity constraints and updatable views that can be used repeatedly in different projects. It allows us to facilitate their implementation.

It is possible to implement data structures that are involved in a constraint in more than one way. Therefore, each constraint type has zero or more corresponding patterns. Next, we present an incomplete list of categories of candidate database design patterns.

1. Patterns about constraints of generalization relationships. Pokrajac et al. [32] has identified the following types of constraints about specialization: overlapping total, overlapping partial, disjoint total, and disjoint partial.

2. Patterns about constraints of whole-part relationships.


4. Patterns that specify how to record state changes of business objects and how to permit only state changes that are found as the result of system analysis.

5. Patterns that specify how to record history of data changes. Date et al. [33] present solution to this problem by using relational variables for recording current and historical data.

6. Patterns that specify how to prohibit a particular data management operation in a database (see an example of one of these patterns is section 3.2).

7. Patterns that specify how to generate surrogate key values.

8. Patterns that specify how to implement updatable views in case of different relational operators that are used in the subquery of a view.

We note that many of the strategies include templates for creating data structures (like base tables).

### 3.2. An example of a database design pattern

**Name:** Deletion of a child depends on its parent  
**Status:** Candidate pattern  
**Data model where it is applicable:** ORSQL  
**Problem:** How to ensure that we cannot delete a row \( r \) from \( \text{Child\_tab} \) if its associated row in \( \text{Parent\_tab} \) satisfies the Boolean expression \( bx \)?

**Related patterns:** A pattern, the use of which ensures that we cannot delete a row \( \text{r} \) from \( \text{Parent\_tab} \) if \( \text{r} \) satisfies the Boolean expression \( bx \).

**Example:** Let us assume that we need to record in a database orders and their lines. We have base tables \( \text{Orders} \) (\( \text{Parent\_tab} \)) and \( \text{Order\_lines} \) (\( \text{Child\_tab} \)) in an ORSQL database (see Figure 2). Table \( \text{Orders} \) has a column \( \text{order\_state\_type} \) (\( \text{cond\_col} \)). We have a rule that we cannot delete the lines of an order, the state of which is "accepted" or "archived" (the corresponding row in \( \text{Orders} \) satisfies the Boolean expression \( \text{order\_state\_type} \text{IN} (2,4) \)).

![Figure 2. A precondition of the use of the pattern](image-url)

<table>
<thead>
<tr>
<th>Parent_tab</th>
<th>PK</th>
<th>parent_tab_pk</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond_col</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Child_tab</th>
<th>PK</th>
<th>child_tab_pk</th>
</tr>
</thead>
<tbody>
<tr>
<td>FK1</td>
<td></td>
<td>child_tab_fk</td>
</tr>
</tbody>
</table>
4. On the pattern-based code-generation

In this section, we present a pattern-based and template-based database code generator. Currently, we have implemented a prototype of the code-generation system by using Visual Basic for Applications. The repository database of the system is implemented by using the desktop DBMS MS Access (2003).

The system supports three types of actors (roles) – pattern describer, pattern manager, and database designer. The main task of a pattern describer is to specify patterns and strategies. The main tasks of a pattern manager are to review patterns and their associated strategies and to decide whether it is time to make these specifications public. It is not possible to publish a strategy before its corresponding pattern is published. On the other hand, each strategy of a pattern can be published in different time after the pattern is published. A pattern manager can also decide that a pattern or a strategy needs improvement and until that must not be accessible to the public. The main task of a database designer is to use the system in order to find suitable patterns, select strategies, and generate code.

How can a database designer use the system in order to generate code? A pattern is a family of solutions [7]. Selection of a pattern is followed by the adaptation of this pattern to a particular context. A designer has to select a strategy from the set (family) of possible strategies and give values to parameters of templates during the adaptation. A designer can check, based on the generated evaluation statements, whether he/she correctly understands the selected pattern / strategy and whether the parameters have correct values. A designer can generate code based on the selected strategy.

Next, we explain informally the concepts of the code-generation system. A database design pattern is a kind of pattern. Each database design pattern uses the terminology of one or more data models (for example, ORSQL) and specifies zero or more strategies. Each database design pattern that specifies one or more strategies can be called as generative database design pattern. Each strategy can be used by the system as the basis of code generation. The result of each code generation is an ordered set of statements that are written in one or more database languages. Each strategy has zero or more preconditions that must be fulfilled in order to use this strategy. For example, a precondition could be that a base table, for which we want to generate a constraint, must have a candidate key that is comprised of a single column (simple key). Another example is that a column, for which we want to generate a constraint, must have VARCHAR type.

Each strategy is characterized by exactly one natural language (for example, English). All the identifiers of design elements (for example, base tables, views in ORSQL) that are created based on this strategy are in this language. In addition, all the templates of evaluation statements and all the preconditions use this language. Each strategy has one or more associated platforms. A platform is a version of a DBMS (for example, PostgreSQL 8.1). Each version of a DBMS has exactly one underlying data model. For example, the underlying data model of PostgreSQL 8.1 is ORSQL. Each version of a DBMS allows us to use one or more languages in order to implement and use databases. For instance, PostgreSQL 8.1 allows us to use declarative SQL and in addition write functions by using procedural languages like PL/pgSQL.

Each platform determines zero or more design element types, the corresponding elements of which can be used in each strategy that is based on this platform. For instance, if the platform is PostgreSQL 8.1, then examples of design element types are "base table", "primary key constraint", and "trigger". Each design element type is specified in zero or more standards. Each standard is used as a basis of zero or more database languages. An example of relevant standard is SQL:2003 [6].

Each strategy consists of one or more statement templates. The templates are ordered within each strategy because generated statements must be read or executed in a certain order. Each template is either an evaluation statement template or a database language statement template. An evaluation statement that is created based on a template is either formal or informal. Each evaluation statement is used in order to describe the particular strategy and its associated pattern. A formal evaluation statement is written in a mathematical language (like the language in [21]). An informal evaluation statement is written in a natural language. A natural language is less precise but hopefully more understandable to the general public than a formal language. We do not prescribe the syntax of evaluation statements. A designer can use these statements for self-evaluation and as comments of the generated code. It is possible to generate code without the existence of evaluation statement templates.

Each strategy is implemented by using one or more languages. A database language statement template is either a create-template or a drop-template. Each create-template is used in order to generate exactly one statement for creating one or more design elements. For example, we can use a CREATE TABLE statement in order to create a base table as well as the primary key constraint of this table. A trigger template is a kind of create-template. It is used in order to create a trigger procedure. Each trigger procedure is
characterized by exactly one execution model (for example, on involved table, on involved columns) [21].

If a create-template is associated with a drop-template, then the latter can be used in order generate a statement for dropping design elements that are created based on the create-template. Drop statements must be presented in reverse order to create statements. Each database object depends on zero or more other database objects and a DBMS checks and preserves these dependencies. If an object o2 depends on object o1, then we firstly have to drop o2 and after that can drop o1.

It is possible that a pattern has more than one strategy for the same platform and natural language. A designer can use the following criteria in order to select exactly one strategy. For example, some criteria allow designers to evaluate reusability of the generated code (1, 2, 4, 5) and performance (4, 6, 7):
1. Preconditions of the strategies.
2. The amount of statements in order to implement strategies.
3. The amount of statements for creating design elements.
4. Whether the strategy uses only declarative constraints or whether it also uses trigger procedures that are created by using an imperative language. Lloyd [34] describes the advantages of declarative languages compared to imperative languages, which include easier teaching, clearer semantics, improved programmer productivity, and better support to meta-programming and parallelism. In current ORDBMS SQL[5] this criterion is rather theoretical because these systems provide only limited support to the creation of declarative constraints [20, 35].
5. Whether the strategy uses only design elements, the types of which are specified in a standard or whether it also uses unstandardized features of a particular platform. For example, PostgreSQL 8.1 allows us to create rules and constraint triggers that are not specified in the SQL standard [6].
6. Execution models of trigger procedures.
7. Whether the generated code contains statements for affecting physical level of a database (for instance, creation of indexes or partitions of base tables) in order to improve performance of data management operations.

Each strategy has zero or more associated parameters. These parameters are used in the statement templates and must be replaced with the actual values, which are provided by a database designer, during the code generation. Some of the parameters are set-valued parameters. Each possible value of a set-valued parameter is a set of values (which is themselves also a value). Each strategy has zero or more templates where it is necessary to generate separate statement based on each element of a value of a set-valued parameter. For instance, we could specify that possible value of a parameter is a set of column names and that it is necessary to generate a separate CREATE TRIGGER statement based on each column name.

A derived parameter is a kind of parameter. A database designer does not give directly a value to the derived parameters during initiation of code generation. Instead, the value of each derived parameter is calculated based on the value of exactly one parameter that is not derived. For example, a strategy that describes how to implement updatable views could have a parameter pr, the possible value of which is a set of column names of a base table. Each of these columns has a corresponding column in a view. A possible value of this parameter is: "empno; salary".

Let us assume that the strategy proposes the use of rules in order to implement updatable views in a PostgreSQL database. The strategy has also a derived parameter, the value of which is calculated based on the current value of pr. In case of the present example the value of this derived parameter would be "NEW.empno, NEW.salary". As you can see, each column name has the prefix "NEW." and the mark ";" is replaced with the mark ",". The word "NEW" can be used in insert and update rules in order to refer to a new row.

4.1. An example of a strategy

Let us assume that the platform is PostgreSQL 8.1. A strategy of the pattern "Deletion of a child depends on its parent" (see section 3.2) contains one create-template and has five parameters. The natural language of this strategy is English.

The precondition of the use of the strategy is that the primary key of the parent table must be simple key.

The parameters of the strategy are the following:
- <PARENT_TAB> - name of the parent table;
- <CHILD_TAB> - name of the child table;
- <PARENT_TAB_PK> - name of the column of the parent table that belongs to the primary key;
- <CHILD_TAB_FK> - name of the column of the child table that belongs to the foreign key that references to the parent table (to its primary key);
- <BX> - Boolean expression.

Secondly, we present an informal evaluation statement template of this pattern: "The deletion of a row in <CHILD_TAB> is not allowed if its associated row in <PARENT_TAB> satisfies the Boolean expression <BX>".

Thirdly, we present the create-template for creating a database rule that implements the pattern:
"CREATE OR REPLACE RULE <#r_<CHILD_TAB>_prohibit_del</#> AS ON DELETE EXISTS (SELECT 1 FROM <#><PARENT_TAB></#> WHERE <#><PARENT_TAB_PK></#> = OLD.<#><CHILD_TAB_FK></#> AND <BX>) DO INSTEAD NOTHING;"

A pattern describer can use zero or more pairs of the special tags "</#>" and "<#>" in each template in order to mark identifiers that are dynamically constructed during the generation of statements. A version of a DBMS can have restrictions to the maximum length of identifiers. The system checks all the generated identifiers that are between the tags. If the length of an identifier is longer than the maximum permitted length, then the system shortens it in order to make it conform to the rules of the DBMS.

Currently, we have developed 24 strategies of 20 patterns based on PostgreSQL 8.1 as the platform. In average, there are 3.4 create-statement templates and 4.4 parameters in a strategy. 9 strategies use user-defined functions that are written in an imperative programming language. 10 strategies use one or more design elements (rules, constraint triggers, or explicit table locking) that are not specified in the SQL standard [6].

4.2. Discussion

The idea of using template-based code generators is not new. This idea has been proposed in the research literature [7] and implemented in the existing systems [36]. What are the advantages of the proposed system?

The proposed and implemented code-generation system is database-based and therefore has all the advantages that a database-based system has compared to file-based systems [19]. It also makes it easy to implement as a web-based system.

The proposed system can be used for generating code for different platforms. Therefore, it is more widely usable compared to the RuleGen framework [21] that aids in implementing data integrity rules in a database that is created by using Oracle DBMS. The proposed system is more than a trigger generator [35], because its generated code is not limited with trigger procedures. For instance, it can generate statements for creating base tables, views and declarative constraints.

Database designers can search strategies based on database design patterns.

The system generates evaluation statements. They help a database designer to make sure that he/she understands the selected pattern and strategy correctly.

The system is designed specifically for using it in the context of database design. For instance, a database designer can select a strategy based on information about indexes or execution models of triggers.

Pattern describers have to declare what the system has to present as the result of code generation instead of specifying the generation procedure by using an imperative language like in [36].

The parameters that are used in the proposed system correspond to the lexical parameters in the MacDonald et al. [7] classification. They allow designers to specify syntactic structure in the generated code. In addition, designers can influence the lexical aspects of the generated code by determining the natural language that is used in the identifiers. In addition, MacDonald et al. [7] propose to use design, performance, and verification parameters in order to influence the result of code generation. In our system, it is possible to influence the structure and performance by selecting one strategy from the set of possible strategies.

In our view, the proposed system has different purposes than object-relational (O/R) mapping tools. O/R tools allow developers to create an object-oriented interface to relational data at the application level. This interface converts data between incompatible type systems. On the other hand, the proposed system allows designers to improve a database schema and its associated behavior according to patterns. It permits the creation of systems that use the features of database systems as much as possible in order to reduce the burden of applications.

5. Conclusions

Object-oriented design patterns are well known to the software engineering community. Unfortunately, database design patterns have received less attention. In this paper, we proposed a pattern-based database design and implementation approach. Preconditions of the use of such approach are the existence of a set of domain-independent database design patterns and the existence of a code generation system that allows database designers to generate code based on the selected patterns. Implementational aspects of patterns are specified in terms of strategies. In this paper, we proposed a set of categories of database design patterns. We presented a template-based and pattern-based code generator.

Future work includes development of pattern languages of the design of object-relational (OR) databases. We need at least one language for each OR data model approach. We also have to implement the code-generation system as a web-based system that uses the help of an ORDBMS SQL and work out more strategies for the existing database design patterns.
6. References


