A Pattern Language for Developing Database-Driven Applications

Osama Mabrouk Khaled
The American University in Cairo
P.O. Box 11431 - 7A Cornich El Nil - Maadi, Cairo, Egypt
+20105800582
okhaled@aucegypt.edu
osama.mabrouk@vodafone.com

Hoda M. Hosny
The American University in Cairo
P.O. Box 2511113 Sharia Kasr El Aini, Cairo, Egypt
+2027975314
hhosny@aucegypt.edu

ABSTRACT

Database-driven applications are some of the most widespread applications nowadays. It normally requires a good knowledge and experience in application development, database SQL, and may be some other fields to build a well-designed application. Our pattern language works on SQL statements as a powerful tool to develop database driven applications. It clarifies some techniques that can lead to a better application development process. All presented patterns are SQL-centric and work on some of the common issues that software developers may encounter.

INTRODUCTION

In 1970, Dr E.F.Codd of IBM published his theory of relational databases in the Communications of the ACM [2]. This work became the mathematically rigorous theoretical base of the relational databases. In 1974, he complemented his theory by publishing the basic twelve principles of relational databases at the Third Annual Texas Conference on Computing Systems. IBM initiated a project called System/R as a prototype which ended in 1974. This project showed the viability of Dr. Codd's theory and a database query language was invented as the first interface language in this domain [2].

The structured query language (SQL) is now the most widely used interface for relational databases. SQL is a descriptive language; users describe in SQL what they want to be done, and the SQL language compiler automatically generates a procedure to navigate the database and perform the desired task. All famous database management systems like DB2, Oracle, Microsoft SQL, Informix, and Sybase use it. They all have common standards even though there are differences from one system to the other. Hence, understanding the standard SQL facilitates the management of database systems. SQL is a simple, yet powerful, database access language.

Applications became more dependent on relational databases as they became the primary source for saving and retrieving information. A database-driven varies in its complexity. Some applications handle small databases while others work with very large databases. For example, an application for maintaining information about employee's absence history is considered small. On the other hand, applications that handle billing information for telecommunication companies are considered quite complex. SQL statements can be constructed in a very complex and parameterized manner inside the code.

This pattern language gives solutions to some of the common problems in database-driven applications. The pattern language provides solutions that can be applied to all database-driven applications. If the pattern language is followed precisely, the maintenance process will be smooth and easy. However, some of the patterns could be applied independently. This pattern language is extracted from a thesis work by the first author for performance optimization in database-driven web applications [4]. Every pattern is organized according to the design pattern template mentioned by Gamma et al [1].

PATTERN LANGUAGE STRUCTURE

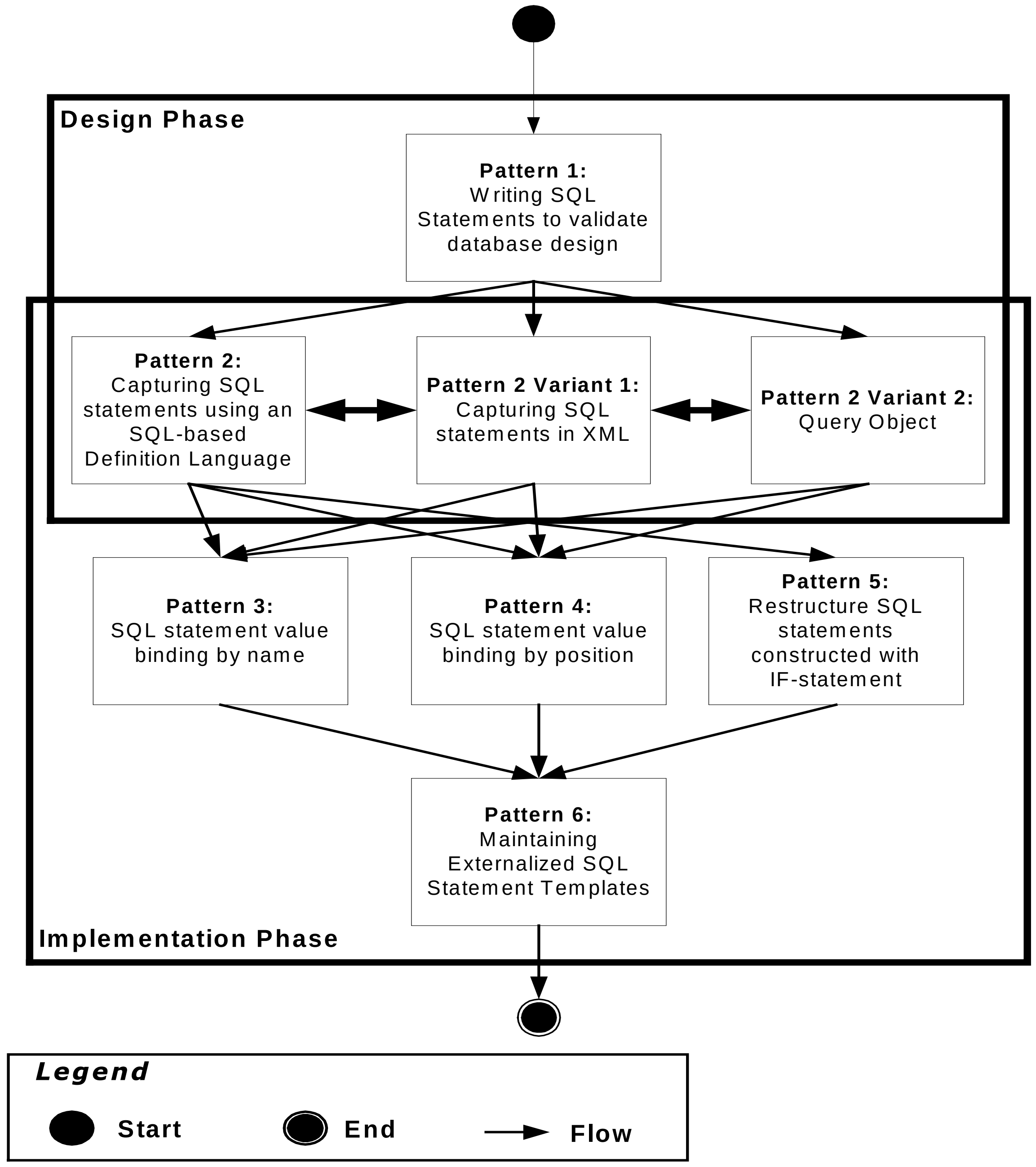
Design Phase
Pattern 1: Writing SQL Statements to validate database design
Pattern 2: Capturing SQL statements using an SQL-based Definition Language
Pattern 2 Variant 1: Capturing SQL statements in XML
Pattern 2 Variant 2: Query Object
Pattern 3: SQL statement value binding by name
Pattern 4: SQL statement value binding by position
Pattern 5: Restructure SQL statements constructed with IF-statement
Pattern 6: Maintaining Externalized SQL Statement Templates
Implementation Phase
Legend
Start
End
Flow

Figure 1 Pattern Language Structure

The pattern language shown in Figure 1 introduces a structured approach to tackle problems one at a time. It highlights the use of the SQL statements in both the design and implementation phases. All these patterns are SQL-centric. They consider the SQL statements as the main point on which the solutions are based. All discussed problems exist in the software domain with different shapes and degrees. The introduced solutions give the best practices regarding their contexts. The pattern language is the result of a wide experience with database-driven applications. The implementation of the patterns is not difficult and can be applied on different platforms. Moreover, the concept can be applied over other backend systems that are built over descriptive commands.

The pattern language introduces two patterns for the development process, another two to organize the externalised SQL statements, and three patterns for implementation issues. Pattern 1 and 6 give Roles &

Responsibilities for both the designer and the implementer. Pattern 2 shows how to capture SQL statements as templates using a well-formed definition language. Pattern 2 has two other variants. The first one is based on the Data Access Object pattern (DAO) introduced by SUN [6] and helps capturing SQL statements using XML. The second variant shows how to capture SQL queries using a pattern called Object Query [3]. This pattern could be part of the design or the implementation phase as shown in Figure 1. Patterns 3 and 4 show how to construct parameterised SQL statements. They are very similar two each other and are variants for the same problem. Finally, Pattern 5 shows how to restructure complex SQL statements with different parts constructed using IF-statements to make them readable and understandable. This pattern can be used also to avoid future code that may depend over IF statements to build SQL statements. This pattern can be viewed also as an anti-pattern.

Pattern 1: Writing SQL Statements to validate database design

Problem

How to validate database design during the design phase?

Context

- The developed application is database-driven.
- Database design validation is needed in the design phase.
- Database design is complete.

Solution

One person cannot complete a building, and one person cannot do everything alone. The architecture engineer has to come out with a design blueprint, and then the civil engineer applies the design. The same is done in software engineering where the software designer generates the artifacts and the implementer applies them. This pattern shows the additional R&R for the database designer. This solution builds upon RUP (Rational Unified Process) by extending the model in order to make working with SQL Statement Template Design Pattern more robust [5].

Figure 2 shows that the activity of building SQL Statement templates must proceed directly after the database design is complete as the team has fresh knowledge of the database architecture. If they combine their knowledge of the database architecture with the application's functional and non-functional requirements, they would reach most of, if not all, the required SQL statements that would be needed in the implementation phase. Practicing such an activity helps in refining the database design as well. Postponing the activity of constructing the SQL Statement templates to the implementation phase would lead to a delay in the development as a whole because the development team has to wait for some time to understand the database architecture before they start. It is not even guaranteed that they can reach as good SQL Statement Templates as the database designers can.

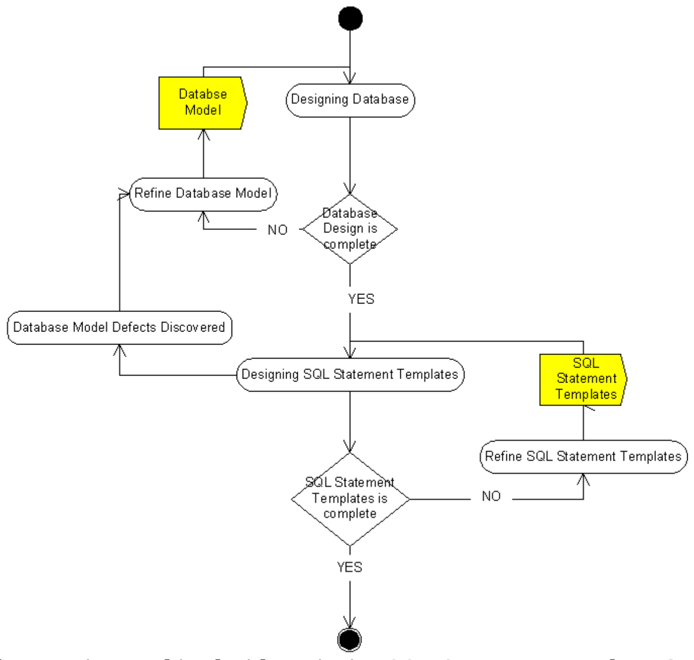
Databse Model
Designing Database
Refine Database Model
NO
Database Design is complete
YES
Database Model Defects Discovered
Designing SQL Statement Templates
SQL Statement Templates
Refine SQL Statement Templates
SQL Statement Templates is complete
NO
YES

Figure 2 Database Design combined with Designing SQL Statement Templates Activity Diagram

For database driven applications, the database design must be introduced in the design phase. The database design task is mandatory because the application database access layer will be built over it. Any design activity in general takes quite some time of discussion in order to arrive at the best shape of the final product. The team that spent its time building a database design would definitely have acquired excellent understanding of the database architecture because they are the ones that took the design decisions. Hence, they are the best candidates to continue building the SQL Statement templates. Figure 3 highlights the main responsibilities for the Database designer and shows the deliverables to the implementation phase.

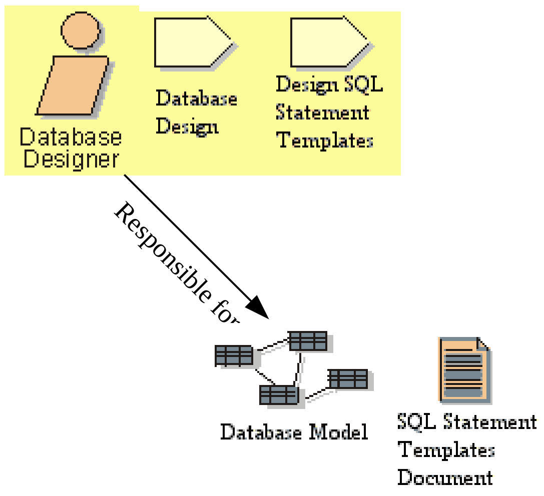
Database Designer
Database Design
Design SQL Statement Templates
Responsible for
Database Model
SQL Statement Templates Document

Figure 3 Database Designer Responsibilities

Forces

- **Development Progress**: consider the impact of applying the pattern on the overall progress of the development process.
- **R&R:** consider the impact of applying the pattern on the responsibilities of the database designer's predefined tasks.
- **Project Size:** consider the impact of applying this pattern regarding the project size.
- **Database Quality:** consider the impact of applying this pattern on the quality of the database design.

Consequences

- The Database designer may be inclined to extend the process of writing SQL Statements and enhancing them which may delay other processes in the development phase. All needed SQL statements may not be acquired during this phase. The purpose is to have as good SQL statements as possible given the timelines.
- Some SQL statements may show problems when they are integrated with application code because if they were not tested against the database structure.

Pattern 2: Capturing SQL statements using an SQL-based Definition Language

Problem

How to capture SQL statements using an SQL-based definition language?

Context

- Pattern 1 is already applied and the templates for the SQL statements are defined.
- There is a need for a SQL statements knowledge base.
- SQL statements knowledge base will be used to build test cases.
- Database model maintenance and support is considered to be easier if all access points are known exactly.

Solution

This solution depends on the comment tag of the SQL "--". It binds the comment tag with special keywords to define the templates. For example, Table 1 defines basic keywords that can be used to capture an SQL statement.

Table 1 SQL Statement Template Definition Tags

Tag	**Tag Name**	**Description**
--@begin NAME	SQL template begin	- Identifies the start of the template - Gives a name to the template - Name must be unique
--@end NAME	SQL template end	- Marks the end of the template and name. - Nested SQL templates are not allowed.
--@group NAME	Template group name	If the template is not assigned to a group it will be added to the default group **default.**
TAG_NAME	Replaceable tag value	- This identifies a replaceable tag at runtime. - Line breaks are not allowed in between
--TAG_NAME	Replaceable comment tag value	This identifies a replaceable comment which is used to enable a certain construct at runtime. The tag must start with --$ and ends with $ with no line breaks in between.

The mentioned definition language in the above table provides a very simple and straightforward technique to capture any kind of SQL statements. The captured SQL statement, which could be defined as "SQL Statement Template", is constructed with all possible parameters that can pass to it. The captured SQL statements could be externalized outside the application code to maximize its benefits. SQL statement templates can be organized in **.sql** files in order to make their accessibility easier. For example, an ideal .**sql** file can be constructed in this way: -

```
--@begin QUERY_1
--@group GROUP_1
```

```
-- This is the comment of the query distributed on more than one line
-- another comment
SELECT * FROM TABLE1 WHERE FIELD1 = $PARAM1$ AND FIELD2 = $PARAM2$
--@end

--@begin QUERY_2
--@group GROUP_1
-- This is the comment of the query distributed on more than one line
-- another comment
SELECT * FROM TABLE2 WHERE FIELD1 = $PARAM1$ AND FIELD2 = $PARAM2$
--@end
.
.
.
--@begin QUERY_X
--@group GROUP_X
-- This is the comment of the query distributed on more than one line
-- another comment
SELECT * FROM TABLEX WHERE FIELD1 = $PARAM1$ AND FIELD2 = $PARAM2$
--@end
```

Variant (DAO)

SQL statements can be captured using XML as done in DAO (Data Access Object) pattern [6]. DAO adopted XML to externalize SQL statements in order to reduce similar APIs that use the same SQL statements.

Variant (Query Object)

It is an interpreter object mentioned by Martin Fowler in [3] that contains other objects which can be formed to generate SQL queries. The query object hides the knowledge of the SQL inside objects in order to get the change in the SQL statements done in one place only [3]. This variant pattern is valid only in cases of SQL queries.

Forces

- **Development Progress:** consider the impact of applying this pattern regarding the overall progress of the application.
- **Maintenance:** DB experts, who are not programmers, will be involved in maintaining the SQL statement templates.

Consequences

This process may lead to a delay in the over all progress of the development process. It may be an element of delay for small projects which do not anticipate to change their SQL statements often.

Pattern 3: SQL statement value binding by name

Problem

How to pass runtime parameters to SQL statements inside the application?

Context

This pattern can be best used in the following situations: -

1. SQL statements pass parameters from the application code. Parameters can be found in the INSERT, UPDATE, DELETE, or SELECT statements, or any other SQL statement called from the application. Parameters can be part of the "where condition", values to be inserted, or values to be updated. These parameters do not corrupt the SQL statement structure. However, their types must be correctly submitted at runtime in order not to get SQL errors.
2. It is possible that some SQL statement parameters take the same value at runtime
3. Some SQL statements have multiple parameters.

Solution

To understand this pattern, it is very important to understand the classical solutions for passing parameter values to SQL statements. One of these solutions is to concatenate the parameter values to the SQL statement. For example, in a Java application, concatenation is achieved in this way

```
"SELECT * FROM INVOICES WHERE INVOICE_TYPE ="+par1+" AND PRICE > "+par2
```

This solution is very straightforward and easy. However, it may lead to an unreadable statement if it is getting a very complex structure.

Another solution, is to have a prepared statement with the parameter placeholders marked with "?". This solution allows the developer to mark the positions of the runtime values and to have at the same time a readable and understandable SQL statement. It binds value by position. "PreparedStatement" uses this placeholder markup technique. They are supported by the Database Management Systems and many programming languages like Java and C++ provide their APIs as well.

The suggested solution follows the same track of the "PreparedStatement", but it uses names for the placeholders inside the SQL statement. For example,

```
SQLStmtTemplate sqlBlock = new SQLStmtTemplate("SELECT * FROM INVOICES WHERE
INVOICE_TYPE = $inv_type_param$ AND PRICE > $price_param$");

sqlBlock.setLong("$inv_type_param$", inv_type);
sqlBlock.setLong("$price_param$",price);

// continue the normal work either by passing it to a prepared
// statement object or to normal Statement handler
```

If we assume that **`inv_type`** was **1** and **`price`** was **2** then the final SQL statement will be

```
SELECT * FROM INVOICES WHERE INVOICE_TYPE = 1 AND PRICE > 2
```

SQL rules does not prevent switching the position of the where conditions (**`INVOICE_TYPE = 1)`** and (**`PRICE > 2),`**as follows, and there will not by any change in the code that binds names with values

```
SELECT * FROM INVOICES WHERE PRICE > 2 AND INVOICE_TYPE = 1
```

`SQLStmtTemplate` is a class that has methods to substitute the correct placeholder type at runtime like "PreparedStatement"

```
class SQLStmtTemplate {
   String sqlStmt;
     public SQLStmtTemplate(String newSQLStmt) {
          this.sqlStmt = newSQLStmt;
     }
     public void setString(String paramName) {
     }
     public void setInt(String paramName) {
     }
     public void setDouble(String paramName) {
     }
      .
      .
      .
     public void setObject(String paramName, int type) {
     }
     public String toString() {
     }
 }
```

The following points summarize the benefits of using placeholder names introduced by this design pattern:-

1. **Definition Semantics:** “Value binding by name” gives semantics to the placeholder which is inherently available if proper names are used with the SQL Statement. Moreover, different placeholders may repeatedly take the same value. In this case, it will be useful to use one name for these placeholders to indicate that they always take the same value. Another benefit is that it provides a dictionary of terminologies that can be shared inside the development team. This technique facilitates very much the knowledge transfer and guarantees no confusion when SQL statements are discussed.
2. **Implementation Complexity:** “Binding value by name” allows the developer to bind by the placeholder name regardless of its position, which is more convenient and understandable. This is definitely more flexible as SQL statements can be changed without any code modification. If the SQL statement has more than one placeholder that takes the same value, then giving these placeholders the same name allows the developer to optimize the written and generated code. Moreover, it shows very clearly that such placeholders take the same value.

Forces

1. **Understandability:** Consider the impact of “parameter name”, as a placeholder, on the understandability of the SQL statement.
2. **Complexity**: Consider the impact of using “parameter name“ as a placeholder on the complexity of the newly developed applications or reengineered ones that have “?” as a placeholder.
3. **Extensibility**: Predefined types must be fully recognized by the design pattern solution.
4. **Learning Curve**: Consider the impact of learning “binding value by name” technique on the overall implementation.
5. **SQL Buildup responsibility:** Consider the impact of using “parameter names” on the consistency of the SQL statements if a different actor builds them and their structure is applicable to modifications.
6. **Code Revisit:** Consider the impact of changing the SQL statement structure, given that the same placeholder names are maintained, without revisiting the code that binds value at runtime.

Consequences

1. It may happen that the SQL statement contains by chance one or more of the parameter names as part of the text, which may lead to corruption in the final SQL statement. So, smart “binding value by name” solution should be guaranteed if this case is applicable.
2. If the implemented application iterates a lot on the SQL statements that use “placeholder names”, then string manipulation may degrade the performance and affect application memory according to the selected platform. For example, intensive string manipulation in java can affect the performance and consequently slow down the response for other requesters.

Pattern 4: SQL statement value binding by position

Problem

How to pass runtime parameters to SQL statements inside the application?

Context

This pattern can be best used in the following situations: -

1. SQL statements pass parameters from the application code. Parameters can be found in the INSERT, UPDATE, DELETE, or SELECT statements, or any other SQL statement called from the application. Parameters can be part of the "where condition", values to be inserted, or values to be updated. These parameters do not corrupt the SQL statement structure. However, their types must be correctly submitted at runtime in order not to get SQL errors.
2. Some SQL statements have multiple parameters.
3. There are standard libraries that help to bind SQL statement values by position.

Solution

This pattern is considered a variant of Pattern 4 where SQL statement uses placeholders marked with "?". This solution allows the developer to mark the positions of the runtime values and to have at the same time a readable and understandable SQL statement. It binds value by position. "PreparedStatement" , which supported by the Database Management Systems and many programming languages like Java and C++, uses this placeholder markup technique.

The suggested solution follows the same track of the "PreparedStatement", but it uses names for the placeholders inside the SQL statement. For example,

```
SQLStmtTemplate sqlBlock = new SQLStmtTemplate("SELECT * FROM INVOICES WHERE
INVOICE_TYPE = ? AND PRICE > ?");
sqlBlock.setLong(1, inv_type);
sqlBlock.setLong(2,price);
// continue the normal work either by passing it to a prepared
// statement object or to normal Statement handler
```

If we assume that **`inv_type`** was **1** and **`price`** was **2** then the final SQL statement will be

```
SELECT * FROM INVOICES WHERE INVOICE_TYPE = 1 AND PRICE > 2
```

`SQLStmtTemplate` is a class that has methods to substitute the correct placeholder type at runtime like "PreparedStatement"

```
class SQLStmtTemplate {
  String sqlStmt;
    public SQLStmtTemplate(String newSQLStmt) {
         this.sqlStmt = newSQLStmt;
    }
    public void setString(String paramName) {
    }
    public void setString(int paramPos) {
    }
    public void setInt(String paramName) {
    }
    public void setInt(int paramPos) {
    }
    public void setDouble(String paramName) {
    }
    public void setDouble(int paramPos) {
    }
     .
```

```
              .
              .
           public void setObject(String paramName, int type) {
           }
           public void setObject(int paramPos, int type) {
           }
           public String toString() {
           }
        }
```
The above class introduces the concept and it does not force the developer to use it as the solution may exist in some commercial libraries.

Forces

1. **Understandability:** Consider the impact of “?”, as a placeholder, on the understandability of the SQL statement.
2. **Complexity**: Consider the impact of using “? “ as a placeholder on the complexity of the newly developed applications or reengineered ones that have “parameter name” as a placeholder.
3. **Extensibility**: Predefined types must be fully recognized by the design pattern solution.
4. **Development Progress**: Consider the impact of applying this pattern on the overall progress of the project.
5. **Code Revisit:** Consider the impact of applying this pattern on the anticipated future changes of the SQL statement structure.

Consequences

If the positions of the placeholders are changed without changing the relevant application code that bind value with position, then unexpected results will be generated out of this code.

Pattern 5: Restructure SQL statements constructed with IF-statement

Problem

How can we get rid of the complex structure of SQL statements that use IF statements?

Context

- A database-driven application with code that generates SQL statements using many IF statements.
- DB Experts cannot easily maintain SQL statements that are written inside the code either because of their ignorance of the programming language or because the code is very complex to the degree that it is not easily understandable or even readable.
- Readability and understandability decreases, as the code that generates the SQL statements gets more complex.
- Externalizing SQL statements using an SQL-based Definition Language Pattern is adopted.

Solution

There are different complexity levels for constructing the SQL statements inside the application code. The simple one does not require replacing anything at runtime. The more complex level is where there are parameters needed to build the SQL statement, but the statement interface is not changed. The most complex level may require existence or non existence of different SQL statement blocks and the statement interface may even change at runtime. The third type is what this pattern tries to solve. Table 2 gives examples of the listed types.

Table 2 SQL Construction complexity

Type	Example

No replaceable value	`SELECT *`
	`FROM INVOICES`
	`WHERE`
	`INVOICE_TYPE = 1`
	`AND PRICE > 30`

Replaceable value only	`SELECT *`
	`FROM INVOICES`
	`WHERE`
	`INVOICE_TYPE = $INVOICE_TYPE$`
	`AND PRICE > $MIN_PRICE$`

Replaceable SQL constructs	`SELECT`
	`--$all$  *`
	`--$invoice_number$ inv_no`
	`FROM`
	`  INVOICES`
	`--$inv_type$ WHERE INVOICE_TYPE = $INVOICE_TYPE$`
	`--$min_price$ WHERE PRICE > $MIN_PRICE$`

The third type listed in Table 2 shows that “field names” and the “where condition” can change at runtime. Application code usually use IF statements to control the structure of the SQL statements. It may look like an unprofessional technique to write SQL statements, but it does happen that developers sometimes write SQL statements in this way to save time, they do not have good experience, or it works only this way. The following simple JAVA code shows how the third example SQL statement can be written using IF statements: -

```
String sqlStmt = “SELECT “;
If (condition1)
   sqlStmt += “ * “;
else if (condition2)
   sqlStmt += “ inv_no ”;
sqlStmt += “ FROM INVOICES where “;
if (condition3)
   sqlStmt += “ INVOICE_TYPE = “+ invoice_type;
else if (condition4)
```

```
          sqlStmt += “ PRICE > ”+ min_price;
```

This code generates actually two SQL statements. It is not easy to grasp the generated SQL statement from the first look although the generated SQL statements are very simple.
This pattern defines the SQL statement in a readable format and makes use of the simple string manipulation, which is usually found in all programming languages. So, the SQL statement can be defined as shown in Table 2. If `SQLStmtTemplate class` ,mentioned in Pattern 4, can be modified to accept extra information.

```
class SQLStmtTemplate {
         String sqlStmt;
          public SQLStmtTemplate(String newSQLStmt) {
               this.sqlStmt = newSQLStmt;
          }
          public void setString(String paramName) {
          }
          public void setInt(String paramName) {
          }
          public void setDouble(String paramName) {
          }
           .
           .
          public void setObject(String paramName, int type) {
          }
          public void clearCommentTag(String commentTag) {
          }
          public String toString() {
          }
}
```

The application code will be ideally similar to this

```
       String sqlStmt = “SELECT \n
       --$all$  * \n
       --$invoice_number$ inv_no  \n
       FROM  \n
         INVOICES  \n
       --$inv_type$ WHERE INVOICE_TYPE = $INVOICE_TYPE$ \n
       --$min_price$ WHERE PRICE > $MIN_PRICE$”  \n “

       SQLStmtTemplate tmplt = new SQLStmtTemplate(sqlStmt);
       If (condition1)
         tmplt.clearCommentTag(“all”);
       else if (condition2)
         tmplt.clearCommentTag(“invoice_number”);
       if (condition3)
         tmplt.clearCommentTag(“inv_type”);
       else if (condition4)
         tmplt.clearCommentTag(“min_price”);
```

Forces

- **Readability**: consider the impact of applying this pattern on understanding the SQL statements.
- **Modularity:** consider the impact of applying this pattern on the modular structure of the application.
- **Development Progress**: consider the impact of applying the pattern on the overall progress of the development process.
- **Human Errors:** consider the impact of applying this pattern regarding the possible human errors that can be generated while modifying the application code.

Consequences

Human errors could happen if the application code is reengineered which may lead to unexpected results. Reengineered code requires to run test scenarios to make sure that the current use cases are working as designed.

Pattern 6: Maintaining Externalized SQL Statement Templates

Problem

How the externalized SQL statements will be maintained? And who will do it?

Context

- SQL Statement Templates are externalized using Pattern 2.
- Integration mechanism between the externalized SQL statements and the application is available.

Solution

SQL Statement Templates that drive the business functionality are generated during the design phase as a separate deliverable. This deliverable can be integrated into the code with the minimum amount of effort. The developers can then easily change SQL Statement Templates in the implementation phase, if required. One or more of the development team can now be responsible for maintaining SQL Statement Template files and for modifying the templates if required (Figure 4). This helps in providing a consistent approach for building the SQL statements and leads the development team to focus more on achieving the functional requirements and not get too involved in database technical issues (Figure 4).

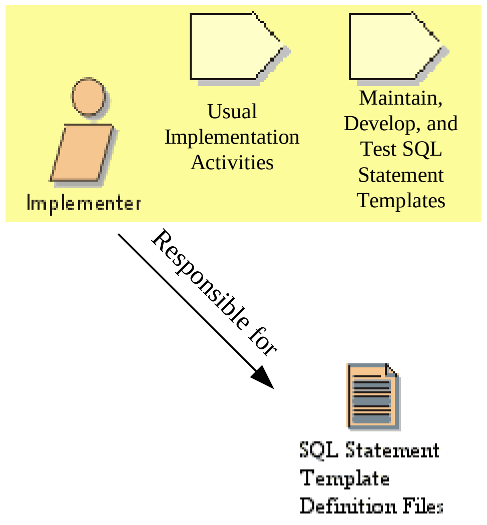
Implementer
Usual Implementation Activities
Maintain, Develop, and Test SQL Statement Templates
Responsible for
SQL Statement Template Definition Files

Figure 4 Implementer responsibility with the SQL Statement Template Definition Files

The number of the SQL Statement Template files will determine the actors who will play the same role in SQL Statement Template maintenance. For example, SQL Statement Templates can be distributed to more than one file according to certain logic. Instead of putting a burden over one developer, Maintenance responsibilities can be distributed among more than one person.

Forces

- **Overall Project awareness:** consider the impact of applying this pattern given that there is at least one person from the development team who is aware of all the functional requirements that lead to all the SQL Statement Templates.
- **Development Progress**: consider the impact of applying the pattern on the overall progress of the development process.
- **R&R:** consider the impact of applying the pattern on the responsibilities of the assigned developer's predefined tasks.

- **Project Size:** consider the impact of applying this pattern regarding the project size.

Consequences

If the SQL Statement Templates are maintained by more than one person, then the approach may be different unless there are guidelines on how to buildup the SQL statement templates.

CONCLUSION

In this paper we presented a pattern language based on six patterns which were also described in the text. The suggested language can be further enhanced to provide more functionality. The Pattern Language in this work is the result of our long experience with database-driven applications. In fact, it is already being used on some currently running systems. The implementation is fairly straightforward and can be applied on different platforms. Moreover, the concept can be easily transferred to other backend systems that are built over descriptive commands. This pattern language is always useful as long as there are separate applications that are communicating with well-known interfaces and which can be changed in the future. Moreover, an application can be developed to help the designer generate the required SQL templates and produce them in the required format.

REFERENCES

[1]. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison. Wesley professional computing series.

[2]. ITWorld.com (2001). *SQL History/Overview*. http://www.itworld.com.

[3]. Martin Fowler and others (2002). *Patterns of Enterprise Application Architecture*. http://www.martinfowler.com/eaaCatalog/.

[4]. Osama Mabrouk Khaled (2004). *Capturing Design Patterns for Performance Issues in Database-Driven Web Applications.* M.Sc. Thesis, Computer Science Department, the American University in Cairo.

[5]. *Rational Unified Process*. http://www.rational.com/

[6]. Sun Microsystems (2002). *Data Access Object (DAO)*. http://java.sun.com/blueprints/patterns/DAO.html.