Self-Generated Information System

Chadi Kallab
Software Architect + IT Internship Mentor
Sabis Educational Services
Adma, Lebanon
chadi@kallab.net

This paper suggests a basic system architecture for a self-generated software that builds itself as per its on-going user requirements. The challenge of this research and its recursive future work is to remove repetitive software development through standardized component building, while offering more and more features without losing sight of performance and flexibility. The idea behind this research was inspired from the existing option in MS Access to allow custom generation of data-entry forms. It initiated as part of a local project Code-Generator to shift focus of development towards more complex features, by reducing that of simple data-entry forms to parameterization. Another part of this research evolved from the architecture of the new Sabis School Management Web-Portal.

Keywords: Self-Generated, Data-Entry, Standards, Automated, Generic, MVC, Model-View-Controller, Regular Expressions.

I. INTRODUCTION

A big majority of software feature development require the repetitive handling (creation, modification or even deletion) of database components (tables, views, user stored-procedures). This is usually followed by a change in code. For those data entry features, the steps to follow are pretty much independent of the data being handled, in a way that it could be parameterized to gain time while developing features that need more complex business rules implementation.

This set of “systematic” features all start at the table definition level. In fact, a database table is defined as a set of fields specified by a (name, type, size ...). Some of these fields may hold values restricted by a foreign key (FK) field from another table. Others may be part of a combination that makes the current record unique. Both unique keys (UK) and primary keys (PK) result in the same effect; but their handling varies.

Once the tables are created, the next step to do in any feature development is to generate the corresponding following database components, as needed: views and user-stored-procedures (USP) to select, insert, update and/or delete data.

The format of any standard select statement, whether inside a view or a USP or simple query, is as follows:

```
SELECT <Field List>
FROM <Table Name>
WHERE (1 = 1)
{AND (<Field Condition>)}
```

The standard format of any insert statement, whether inside a USP or simple query, is as follows:

```
INSERT INTO <Table Name> (<Field List>)
VALUES (<Value List>)
```

The standard format of any update statement, whether inside a USP or simple query, is as follows:

```
UPDATE <Table Name>
SET {<Field> = <Value / Parameter>}
WHERE (1 = 1)
{AND (<Field Condition>)}
```

The standard format of any delete statement, whether inside a USP or simple query, is as follows:

```
DELETE FROM <Table Name>
WHERE (1 = 1)
{AND (<Field Condition>)}
```

The standard format of any view drop & create statement is as follows:

```
IF EXISTS(SELECT * FROM sys.views
WHERE [object_id]
= OBJECT_ID(N'<View Name>')
DROP VIEW <View Name>
GO
CREATE VIEW <View Name>
AS
<Query with no Parameters>
GO
```

The standard format of any USP drop & create statement is as follows:

```
IF EXISTS(SELECT * FROM sys.objects
WHERE [object_id]
= OBJECT_ID(N'<USP Name>')
AND type in (N'P', N'PC'))
DROP PROCEDURE <USP Name>
GO
CREATE PROCEDURE <USP Name>
(
    <Parameter Definition List>
) AS BEGIN
    <SQL Query>
END
GO
```
Each item in the “<Parameter Definition List>” of any USP abides by the format:

\(<Name> <Type> \{(<Size> [, <Precision>])\} [= <Default Value>] \) [OUTPUT]

All parts of the above stated formats, that do not start with a keyword, and included between curly-brackets or end with the term “List” are comma-separated, of course except the first item.

The straight-brackets state that the enclosed part is optional for most cases but required for specific ones.

In parallel, windows application or web forms can be viewed as collection of controls or group of controls with a set of properties defining the identity of the form. Most data entry forms are mapping one or more table with events triggering the call of Select, Insert, Update and/or Delete USPs. These forms are divided into 2 main categories: Multiple Entries and Single Entry.

Multiple entries forms template to a grid of entries with the possibility to add new records, edit or delete existing ones.

The delete option requires no additional data than the ones already available in the grid at the selected index before calling the operation.

The other 2 options usually redirect the user to a blank single entry form for new records, or one with the data of the selected record to update. Once the addition or update is done, the user is redirected to the form that requested the single entry, which usually happens to be that of the multiple entries if exists.

Most single entry forms follow the same pattern of displaying sequentially the fields for the user to enter data in, followed by at least 2 of the following options: Save, Clear and Back to List. Some forms have one more option to “Save and New” to keep entering new records. This option is added mainly to simply clearing the fields after submit instead of changing forms for every new record.

To go back to the multiple entries forms, even some additional features, like record(s) search or column sort, can be mapped to one same algorithm with of course little modifications if needed.

From a management perspective, even when the technical team working on each project has been developing such features for a long enough time to get used to this systematic steps, the question remains whether or not the automation of such features can reduce the development and/or maintenance time cost, while keeping enough flexibility for business changes.

This paper aims to suggest a first version of a system that automates this process based on the user requirement for new data entry features, and, in later versions embeds the self-implementation of more complex features.

II. SYSTEM ARCHITECTURE

In order to automate the process mentioned previously, we need first to clarify what are the main tasks of database then of code management for this kind of features, and how they are related, and flow of execution.

Starting at the database level, we can state the following cases:

1) {Create, Alter, Drop} Table
2) {Add, Alter, Drop} Field
3) {Add, Alter, Drop} Constraint
4) {Create, Alter, Drop} View
5) {Create, Alter, Drop} USP
6) Insert record(s)
7) Select record(s)
8) Update record(s)
9) Delete record(s)
10) Execute USP

It is best practice to arrange the code as per the MVC (Model-View-Controller) code model, which relies mainly on 3 major layers: Data Access Layer (DAL), Business Entity Layer (BEL) and Business Logic Layer (BLL).

The DAL (Data Access Layer) is responsible for pulling from database or pushing into database raw data.

The BEL (Business Entity Layer) maps the tables and/or select results into classes where schema validation is made at the code level.

The BLL (Business Logic Layer) is responsible for verifying business rules on data from either DAL and/or BEL. In addition, this layer converts between raw data from DAL and collections of entities from BEL.

The main steps to follow when coding are:

1) Update Layers (DAL, BEL, BLL)
   a. {Create, Modify, Delete} Class
   b. Add methods for components handling
      - {Create, Alter, Drop}
   c. Add methods for records handling
      - {Select, Insert, Update, Delete}
2) Create Admin Form(s) for components mgt.
   - {Create, Alter, Drop}
3) Create Multiple Entries Form(s)
4) Create Single Entry Form(s)
5) Create Main Form & Connecting Form(s)
6) Create Configuration Items
7) Update Form(s) as per Configuration Items
The diagram below shows the dependencies between the different components of the system, where the thickness of the line represents the strength of the relation which is directed towards the dot.

The standard flow of any select operation is shown as follows:
1) UI requests a set of data from BLL
2) BLL calls DAL for raw data extraction
3) DAL connects to DB and sends request
4) DB returns results to DAL
5) DAL returns raw results to BLL
6) BLL requests validations from BEL
7) BEL returns schema validation to BLL
8) BLL returns results collection to UI

Note that at any level, if an error occurs, a message is returned to the upper level instead of the effective data.

The standard flow of any insert or update or delete operation is shown as follows:
1) UI sends data to BLL
2) BLL requests validations from BEL
3) BEL returns schema validation to BLL
4) BLL calls DAL for operation
5) DAL connects to DB and sends request
6) DB returns result success to DAL
7) DAL returns result success to BLL
8) BLL returns result success to UI

The standard flow of any create or alter or drop operation is shown as follows:
1) UI sends data to BLL
2) BLL calls DAL for operation
3) DAL connects to DB and sends request
4) DB returns result success to DAL
5) DAL returns result success to BLL
6) BLL returns result success to UI

The next step to go through is to make the main database operations mentioned earlier, generic, standard and as complete as possible.

In that line of thought, it is best practice for a table to have one single PK (primary key) of integer type, named after the table and ending with “id”.

This field could be auto-incremented by the SQL engine, or to be on the safer side be the result of the query:

```sql
SELECT ISNULL(MAX(<PK Field>), 0) + 1
FROM <Table Name>
```

The generic standard select procedure would have:
- “Select” either in the prefix or suffix
- All parameters defaulting to NULL
- Parameter Name = “@” + Field Name
- Additional parameters for each field:
  - @selectAll<FieldName> BIT = 1
    - Automatic Select All
  - @cmp<FieldName> BIT = 0
    - Field-Parameter Comparison
  - @<FieldName>_UpperBound <Parameter Type> = NULL
    - If condition is a range

<table>
<thead>
<tr>
<th>@cmp</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Exact</td>
</tr>
<tr>
<td>1</td>
<td>Starts with</td>
</tr>
<tr>
<td>2</td>
<td>Ends with</td>
</tr>
<tr>
<td>3</td>
<td>Contains</td>
</tr>
<tr>
<td>4</td>
<td>In Range</td>
</tr>
<tr>
<td>5</td>
<td>In Subset</td>
</tr>
<tr>
<td>6</td>
<td>Less</td>
</tr>
<tr>
<td>7</td>
<td>Greater</td>
</tr>
<tr>
<td>10</td>
<td>Not Exact</td>
</tr>
<tr>
<td>11</td>
<td>Not Starts with</td>
</tr>
<tr>
<td>12</td>
<td>Not Ends with</td>
</tr>
<tr>
<td>13</td>
<td>Not Contains</td>
</tr>
<tr>
<td>14</td>
<td>Not In Range</td>
</tr>
<tr>
<td>15</td>
<td>Not In Subset</td>
</tr>
<tr>
<td>16</td>
<td>Not Less</td>
</tr>
<tr>
<td>17</td>
<td>Not Greater</td>
</tr>
</tbody>
</table>

The comparison parameter’s value follows the below table:

- Each field condition would satisfy the select all condition (@selectAll<FieldName> = 1) or at least one of the following conditions for @cmp<FieldName>:
  - @<FieldName> = <Field>
  - ToStr(<Field>) LIKE ToStr(@<Field>) + '%'
  - ToStr(<Field>) LIKE '%' + ToStr(@<Field>)
  - <Field> BETWEEN @<Field> AND @<Field>_UpperBound
  - <Field> IN Split(ToStr(@<Field>))
  - <Field> < @<Field>
  - <Field> > @<Field>
  - #N/A
  - @<Field> = #N/A
  - @<Field> != <Field>
  - ToStr(<Field>) NOT LIKE '%' + ToStr(@<Field>)
  - ToStr(<Field>) NOT LIKE ToStr(@<Field>) + '%'
  - ToStr(<Field>) NOT LIKE '%' + ToStr(@<Field>) + '%'
  - NOT (<Field> BETWEEN @<Field> AND @<Field>_UpperBound)
  - <Field> NOT IN Split(ToStr(@<Field>))
  - <Field> >= @<Field>
  - <Field> <= @<Field>

For fields of String types, the equality operator needs first to trim white spaces before comparing values.

Depending on the SQL engine, the “ToStr” method used below would have a different implementation, but with one same purpose. The “Split” method returns a table of values extracted from its input value.

The generic standard insert procedure would have:
- All output parameters defaulting to NULL
- PK field considered output and/or input
- Body as per the below format:
  ```sql
  DECLARE @outTable TABLE
  (<Output Field List>)
  INSERT INTO <Table Name>
  (<Input Field List>)
  OUTPUT
  (INSERTED.<Output Field List>)
  INTO @outTable
  VALUES
  (<Input Parameter List>)
  SELECT {<Output Parameter> = <Output Field>}
  FROM @outTable
  ```

The generic standard update procedure would have:
- PK field not updateable
- Updateable fields defaulting to NULL
- Additional parameter for each updateable field:
  - @consider<FieldName> BIT = 1
    - Consider Field for update
- Condition fields not updateable
- Condition section similar to that of generic standard select procedure.

The generic standard delete procedure would have:
- Condition section similar to that of generic standard select procedure.

One more useful generic procedure would be a save procedure that would have:
- PK field as input and output
- PK field defaulting to NULL
- IF record with specified PK exists
  Update as per PK
ELSE
  Insert and return new PK

To make the full cycle of the system clearer, we need now to discuss the activity sequences behind the different database operations as listed below.

Create Table operation:
1) Specify table name
2) Generate the primary key
3) Create the table

Drop Table:
1) Specify table name
2) Drop constraints with other tables
3) Drop related views
4) Drop related USPs
5) Drop the table

Add Field / Alter Field:
1) Specify table name
2) Specify field name
3) Specify type, size, precision, and if field allows null values
4) Verify values existing or allowed
5) Alter the table

Drop Field:
1) Specify table name
2) Specify field name
3) Drop related constraints
4) Alter the table
5) Alter related views
6) Alter related USPs

Add Constraint
1) Specify table name
2) Specify if UK or FK
3) Specify field name

IF it is a UK Constraint
4) Alter the table
ELSE IF it is an FK Constraint
4) Specify foreign table name
5) Specify foreign field name
6) Alter both tables

Drop Constraint
1) Specify table name
2) Specify if UK or FK

IF it is a UK Constraint
3) Alter the table
ELSE IF it is an FK Constraint
3) Specify foreign table name
4) Alter both tables

Create View / Alter View

To be able to apply these sequence activities, we need to have a basic collection of controls, divided into one or more forms that may be called one after the other if need be:

- Label + Text-Box Database
- Button Set-Database
- Label + Drop-Down-List Components {Table / Field / Constraint / View / USP}
- Label + Text-Box Table-Name
- Button Set-Table-Name
- Label + Panel Field-Properties
  o Label + Text-Box Name
  o Button Set-Field-Name
  o Label + DDL Types
  o Label + Text-Box Size
  o Label + Text-Box Precision
  o Check-Box Allows-Null
  o Label + Text-Box Default-Value
- Label + Panel Constraint-Properties
  o Label + Text-Box Name
  o Button Set-Constraint-Name
Now that we have discussed the admin form(s), the next forms to be used are the Multiple Entries composed as follows:

- **Label + Panel** View-Properties
  - Label + Text-Box Name
  - Button Set-View-Name
  - Label + Panel Join-Properties
    - Label + DDL Direction
      - Is-UK
      - Is FK
    - Radio-Button Is-FK
  - Label + DDL Foreign-Tables
  - Button Set-Foreign-Table
  - Label + DDL Foreign-Fields

- Button Create-New
- Button Delete-Multiple
- Label + Text-Box Search
- Button Go
- Label + Grid-View Entities
  - In-Row Edit
  - In-Row Delete

The third type of forms to be generated by the system is that of single-entry that, due to the variety of field types involved, share one same general (repeater-template) layout:

- Hidden-Field <PK>
- For each field to be entered
  - Label <Field Name>
  - Data Entry Control as per Type:
    1) Integer, No Restriction
      - Text-Box
      - Reg.-Ex. = "-?[0-9]"
    2) Integer, Range Restriction
      - Text-Box
      - Reg.-Ex. = "-?[0-9]"
      - Text-Box + Compare-Validator for Lower-Bound
      - Text-Box + Compare-Validator for Upper-Bound
    3) Integer, Subset Restriction
      - Drop-Down-List
    4) Decimal, No Restriction
      - Text-Box
      - Reg.-Ex. = "-?[0-9].?[0-9]"
    5) Decimal, Range Restriction
      - Text-Box
      - Reg.-Ex. = "-?[0-9].?[0-9]"
      - Text-Box + Compare-Validator for Lower-Bound
      - Text-Box + Compare-Validator for Upper-Bound
    6) Decimal, Subset Restriction
      - Drop-Down-List
    7) Alpha-Numeric, No Restriction
      - Text-Box
    8) Alpha-Numeric, Subset Restriction
      - Drop-Down-List
    9) Date, No Restriction
      - Date-Picker
10) Date, Range Restriction
   ➔ Date-Picker
   ➔ Text-Box + Compare-Validator for Lower-Bound, not before 1671-01-01
   ➔ Text-Box + Compare-Validator for Upper-Bound, not after 9999-12-31

11) Date, Subset Restriction
    ➔ Drop-Down-List

12) Time, No Restriction
    ➔ Time-Picker

13) Time, Range Restriction
    ➔ Time-Picker
    ➔ Text-Box + Compare-Validator for Lower-Bound, not before 00:00:00.000
    ➔ Text-Box + Compare-Validator for Upper-Bound, not after 23:59:59.999

14) Time, Subset Restriction
    ➔ Drop-Down-List

15) Date-Time, No Restriction
    ➔ Date-Picker
    ➔ Time-Picker

16) Date-Time, Date Range Restriction
    ➔ Date-Picker
    ➔ Time-Picker
    ➔ Text-Box + Compare-Validator for Lower-Bound, not before 1671-01-01
    ➔ Text-Box + Compare-Validator for Upper-Bound, not after 9999-12-31

17) Date-Time, Date Subset Restriction
    ➔ Drop-Down-List for Date
    ➔ Time-Picker

18) Date-Time, Time Range Restriction
    ➔ Date-Picker
    ➔ Time-Picker
    ➔ Text-Box + Compare-Validator for Lower-Bound, not before 00:00:00.000
    ➔ Text-Box + Compare-Validator for Upper-Bound, not after 23:59:59.999

19) Date-Time, Time Subset Restriction
    ➔ Date-Picker
    ➔ Drop-Down-List for Time

20) Date-Time, Date Range Restriction, Time Range Restriction
    ➔ Date-Picker

21) Date-Time, Date Subset Restriction, Time Range Restriction
    ➔ Drop-Down-List for Date
    ➔ Time-Picker
    ➔ Text-Box + Compare-Validator for Lower-Bound, not before 00:00:00.000
    ➔ Text-Box + Compare-Validator for Upper-Bound, not after 23:59:59.999

22) Date-Time, Date Range Restriction, Time Range Restriction
    ➔ Date-Picker
    ➔ Drop-Down-List for Time
    ➔ Text-Box + Compare-Validator for Lower-Bound, not before 1671-01-01
    ➔ Text-Box + Compare-Validator for Upper-Bound, not after 9999-12-31

23) Date-Time, Subset Restriction
    ➔ Drop-Down-List

24) Reference, Single Display Field
    ➔ Drop-Down-List
    ➔ Text-Box for Display Field

25) Reference, Multiple Display Fields
    ➔ Grid-View

26) Regular-Expression Based
    ➔ Text-Box
    ➔ Text-Box for Reg.-Ex. Value
    ➔ Reg.-Ex.
    - Button Save
    - Button Save and New
    - Button Cancel

Once all these forms are created, the next logical step to do would be to implement one or more navigation form that will simply redirect the user to the requested multiple-entries form.

The performance factor of the system depends tightly on the technologies used to implement the system. However, based on the available tests on already implemented tools mentioned in this paper, we can deduce that the SQL section of this system do
run at first in the vicinity of the traditional approach. If the system's database is implemented on an SQL engine built to cache views and procedures, then using generic standard versions as suggested would definitely enhance the performance.

As for the code side of the system, the layers would be written in a generic and optimized way to avoid including all properties with default values set in an optimal way, and methods that can be grouped and handled by conditions on arguments. If the user interfaces are built as mentioned in this paper, the display and/or values validation would converge into generic algorithms to perform better.

III. Conclusion

This research tried to suggest a basic system architecture for a self-generated software that builds itself as per its on-going user requirements. It proposed a first version of the part of the system that handles data-entry forms along with its database components, leaving more room for forms and features more complex and challenging to automate, either due to the business rules they may involve, or the complexity of the data being processed.

Proposing standard query formats for the main database components, code and user-interface templates for the data-entry forms would allow, as first attempt, to reduce partly the development cost of new software.

Even though parts of this research are already implemented in many Sabis Software projects, especially in the new Sabis School Management Web-Portal, where latest Microsoft technologies, like auto-generation templates, helped cut considerably the cost of development.

The next research work plan would be, from one side, to integrate the different parts already implemented in a less technical user-friendly way: custom generation of data-entry forms, by reducing the development of simple data-entry forms to parameterization; along with the use of auto-generation templates for database code and user-interface components.

Another research angle would be to investigate about potential patterns that lie behind the process of requirement gathering, of data-entry forms, slightly more evolved features to complex mind-scratching requirements. Of course, the complexity of these features is mainly measured by the difficulty of the business rules they have to follow and/or by the complexity of the user-interface itself.

IV. Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEL</td>
<td>Business Entity Layer</td>
</tr>
<tr>
<td>BLL</td>
<td>Business Logic Layer</td>
</tr>
<tr>
<td>DAL</td>
<td>Data Access Layer</td>
</tr>
<tr>
<td>FK</td>
<td>Foreign Key</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>PK</td>
<td>Primary Key</td>
</tr>
<tr>
<td>UK</td>
<td>Unique Key</td>
</tr>
<tr>
<td>USP</td>
<td>User Stored Procedure</td>
</tr>
</tbody>
</table>

V. References

[1] Microsoft Access Tool