CHAPTER 2

AUTOMATED CODE AND GENERIC FORM GENERATOR

2.1 INTRODUCTION

This chapter proposes a method on automated generic code and form generation. The goals of this chapter are to design and implement an automated generic source code generator from a simple textual class specification and automatic form generation for an existing database.

Gabriel (1994), proposed design patterns as an attempt to capture expertise in building object-oriented software. A design pattern describes a solution to a recurring design problem in a systematic and general way. But beyond a description of the problem and its solution, software developers need deeper understanding to tailor the solution to their variant of the problem. Hence a design pattern also explains the applicability, trade-offs, and consequences of the solution. It gives the rationale behind the solution, not just a part answer. A design pattern also illustrates how to implement the solution in standard object-oriented programming languages like C++ and Smalltalk and facilitates code generation.

Many design tools represent the class diagram and corresponding state models in a textual format. In this chapter the assumption is made such that the class contains not only operation signatures and variable declarations, but also the class behavior is specified by the Extended Finite State
Machinestate model. The goal is to generate fully executable and correct source code as specified by the state model.

Sandra &Sheetal(2002), proposed the idea of automatic form generation - WWW-based input forms implemented as Java applets for updating a WWW-accessible database. Jayapandian (2009) said, creating a forms-based interface for an existing database requires careful analysis of its data content and user requirements. To design structured forms, an interface developer must have a clear understanding of what data is available, its structure and semantics, in addition to predicting user needs. The task of form design, creation and deployment can be very difficult if the schema is large and complex and if the querying needs are diverse. In such scenarios it is worth exploring the extent to which form creation can be automated. Thus our goal is to automate the task of generic form generation and significantly reduce, if not eliminate, the developer’s role in the process.

2.2 CODE GENERATOR FROM DESIGN

The proposal of the code generator from the design is give in Figure 2.2. The code generator prompts for the name of a file that contains the class specification as shown in Table 2.1 and generates file(s) containing the executable code for the class. The class specification is taken from the EFSM shown in Figure 2.3 generated from the rational tools. Parser is used to get the class specification. The code generator created is tested for the “queue” example. Class diagram for the same is shown in Figure 2.1. The sample code generated and driver is shown in Figure 2.4 and 2.5 respectively.

2.2.1 ASSUMPTIONS TAKEN FOR CODE GENERATOR

1. It is assumed that the code is generated for an object-oriented language, e.g., C++ or Java.
2. The code generator is a passive generator.

3. The code generator should generate syntactically correct source code.

4. The code generator should generate semantically correct source code as specified by the state model.

5. It is assumed that class specification is correct. It means that the code generator does not have to handle class specification errors, e.g., issuing error messages or warnings.

6. The code generator should be general, i.e., it should work for any correct class specification as described in project description.

7. The code generator should generate code only for one class.

8. It is assumed that all operations are public and all attributes are private.

9. It is assumed that the class for which the source code is generated is a stand-alone class and it is not related to any other class, e.g., by inheritance, aggregation or association relationship.

<table>
<thead>
<tr>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>int I</td>
</tr>
<tr>
<td>int a[10]</td>
</tr>
<tr>
<td>add(int x)</td>
</tr>
<tr>
<td>int remove()</td>
</tr>
</tbody>
</table>

*Figure 2.1 Class queue*
Figure 2.2 Proposal of a code generator from design

Example

Figure 2.3 EFSM for Class queue
Table 2.1 Class Specification

<table>
<thead>
<tr>
<th>CLASS</th>
<th>queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPERATIONS</td>
<td>void add(int x); int remove();</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>int i; int a[10]; int max;</td>
</tr>
<tr>
<td>STATES</td>
<td>start empty partial full</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TRANSITION LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSITION</td>
</tr>
<tr>
<td>TRANSITION</td>
</tr>
<tr>
<td>TRANSITION</td>
</tr>
<tr>
<td>TRANSITION</td>
</tr>
<tr>
<td>TRANSITION</td>
</tr>
<tr>
<td>TRANSITION</td>
</tr>
<tr>
<td>TRANSITION</td>
</tr>
</tbody>
</table>

END
Note: The class specification is taken from EFSM and it is given as the input to the code generator written using java and the output of the code generator is the complete C++ code of the design. This proposed solution can be integrated to IBM Rational rose. The detailed code for automatic code generation is given in Appendix 1.
2.3 AUTOMATIC FORM GENERATION

A generative algorithm for creating generic input forms on websites from the database tables given as base has been proposed. Most of the forms needed to fill a database on the Internet are hard-coded and require changes in the forms if any changes are needed to be done to the database. This chapter proposes an algorithm and simulation of effectiveness of using generative approach for form generation. Only database needs to be structured and the program shall generate the input form for it and populate the tables by itself, reducing designing incompatibility and overheads. An algorithm to create generic forms, which gets defined at run time through the target database, is proposed. Hence, the forms are consistent with the tables and need not be coded explicitly. Only the inclusion of proposed code in program will generate the forms automatically, based on the specified database. As any web service maintains a server side database to keep records of all its clients and for other management purposes, forms emerge as the basic interaction media for handling the user’s inputs. As the services provided increases, so is the need of generating and maintaining new databases and successively the need of creating new forms increases. The proposed algorithm automates the form creation process and is generic enough to be applied to any service, analysis shows the algorithm is less time consuming and more reliable than conventional methods of form creation. The results have been discussed in later sections.

In any kind of form we have the following types of input areas:

- Simple Text Field
- Radio Buttons
- Checkboxes
- File uploads
• Selection from a List
• Non-editable entries
• Fetching relevant data from other database table (Requires Joining)

For implementing the algorithm, the database scheme needs to be known, also the nature of all its attributes, i.e. their names, types, ranges, etc. shall be known beforehand.

2.4 MODEL STRUCTURE FOR FORM GENERATION

The following models to be implemented for application of proposed algorithm

(i) Entity class is specific class, which contains array variables to store the attribute names, that is of various types. Figure 2.6 portrays the classes and their relations.

(ii) A generic Table Class which serves as a base class for all the derived entity classes that contains:

a) All the array variables present in its derived entity classes, with generic data, which is applicable on all the tables of the database.

b) Functions to retrieve the details about the attribute of a specific table, from the database.

(iii) Display class consisting of following:

a) Functions to form arrays for different type of attributes.
b) Functions to display the form rows without customization (with simple text inputs).

c) Function to invoke the functions to convert the non-customized forms into customized ones as per the attribute types and the styling of the form fields as per the specification.

![Hierarchy of classes for form Generation]

**Figure 2.6 Hierarchy of classes for form Generation**

### 2.5 METHODOLOGY FOR FORM GENERATION

For better understanding of the algorithm, components used are explained and an entity ‘course’ has been considered for which an input form is required. This is done in five steps. The schema of the course table is as shown in Table 2.2.

**Table 2.2‘Course’ Table Schema**

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Description</th>
<th>Code</th>
<th>OwnerID</th>
<th>DeptID</th>
<th>Year</th>
<th>Sem</th>
<th>Type</th>
<th>Image</th>
</tr>
</thead>
</table>


1. An object of the class Entity is instantiated say ‘obj’. This instantiation makes a call to the class ‘Table’. Here the array variables of the entity class are merged with the common variables of the class ‘Table’. In the Object ‘obj’ let there be a variable that contains the attribute names of various types. Therefore, as per the schema, values in different variables are shown in Table 2.3.

**Table 2.3 Types of attributes and corresponding fields in Object**

<table>
<thead>
<tr>
<th>Types</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>searchButtonType</td>
<td>Owner ID [from user], Dept ID [from Dept]</td>
</tr>
<tr>
<td>optionSelectType</td>
<td>Year, Sem, Type</td>
</tr>
<tr>
<td>nonShowType</td>
<td>Id</td>
</tr>
<tr>
<td>uploaderType</td>
<td>Image</td>
</tr>
</tbody>
</table>

2. All table attributes from the database schema are retrieved along with their types by using the following query:

```sql
SHOW COLUMNS FROM {tableName};
```

Each tuple vector contains the following information:

- **Fields**: The value that’s present in the specific column.
- **Type**: The type of the attribute, i.e. integer, real, string, char, etc.
- **Null**: Whether null values allowed or not in the column.
- **Key**: Is the column a primary, foreign or a unique key.
• Default: The default value of the column if no input is given.

• Extra: The extra information associated with the column like comments, etc.

Out of these six vector tuples, only ‘Fields’ and ‘Type’ for the basic format, which are provided in the table 2.4.

Table 2.4 Table attributes and their corresponding types

<table>
<thead>
<tr>
<th>Field</th>
<th>Id</th>
<th>Name</th>
<th>Desc</th>
<th>Code</th>
<th>OwnerID</th>
<th>DeptID</th>
<th>Year</th>
<th>Sem</th>
<th>Type</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Int(10)</td>
<td>Text</td>
<td>Text</td>
<td>Varchar(24)</td>
<td>Int(10)</td>
<td>Int</td>
<td>Int</td>
<td>Int</td>
<td>Var</td>
<td></td>
</tr>
</tbody>
</table>

3. The form is now displayed in a non-customized way i.e.in a plain HTML format consisting of the entire column entries in the table, with each associated to the input type text. Suitable names and ids are given to the inputs to have a better control on them through various client and server based scripts as well as with style-sheets.

Example:

```
<div class='table-row-label' >".Column Name."</div>

<div class='table-row-field ' id='\{COLUMN_NAME\}_input'>
<input type='text' name='\{COLUMN_NAME\}' id='\{COLUMN_NAME\}' value='\${value}' />
</div>
```

The value of COLUMN_NAME will be stored in the ‘field’ column extracted in step 2 and the ‘type’ column can be used to do the validations. The output of this step is as shown in the Figure 2.7.
4. EveryoptionSelectTypes, nonEditTypes, nonShowTypes, radioSelectTypes, checkboxSelectTypes, uploaderTypes, searchColumnType array variables are stored in form of JavaScript arrays, which are then sent to the respective functions for form manipulation.

- showSearchBtn(searchColumns)

This function is used for displaying the search button which will get the required data. For example, if in a form field an ID is required which is an integer, as remembering all the ID’s is not possible. The class entity is joined with the table which contains the association of Names with IDs. Join is used to maintain the integrity of the database.

Thus, for every value in the array search Columns, the corresponding row from the non-customized form is obtained and converted to search input type with the help of client-side scripts, on click functions are assigned to them, so as to pass the following:
- Name of the table that needs to be joined.

- Attribute to be taken from table containing Name-ID association, which are later fetched through the database queries.

- `showSelectBox(optionSelectKeys)`

  This function stores the attribute column which requires a 'select' dropdown menu.

  For every value in the array `optionSelectKeys`, the corresponding rows from the non-customized form are obtained and converted into the Select/Option type, providing options as per the column name.

- `readOnlyNonEditKeys(nonEditKeys)`

  This function stores the attribute columns, which are non-editable. For every value in the array `nonEditKeys`, the corresponding rows from the non-customized form are obtained and converted into non-editable types.

- `hideNonShowKeys(nonShowKeys)`

  This function stores the attribute columns which are hidden from the form. These may be the values that will be automatically filled. For every value in the array `nonShowKeys`, corresponding rows from the non-customized form are obtained and hidden.

- `showUploaderKeys(uploaderKeys)`

  This function stores the attribute columns which are FILE type and needs to be uploaded.
5. After the application of all the above functions, the final form is rendered with all appropriate types of inputs. The form obtained in this step is used to get inputs from the user, for the table. All the input fields are consistent with what is required. The final output is shown in Figure 2.8. The code for the above explanation is given in Appendix 2.

![Final formatted output form](image)

**Figure 2.8** Final formatted output form

### 2.6 COMPARISION BETWEEN HTML CODING AND PROPOSED METHOD

Repetitive analysis was done on a set of 200 forms of varying fields and complexity, the findings were averaged and summarized comparing various aspects of form generation, validation and errors incurred using plain HTML and the proposed method. The findings are shown in Table 2.5.
Table 2.5  Summary of analysis comparing conventional HTML coding with the proposed method

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Estimation Criteria</th>
<th>HTML Coding</th>
<th>Proposed Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Time taken in generating effective lines of code for a form.</td>
<td>5-8 minutes</td>
<td>10-20 seconds</td>
</tr>
<tr>
<td>2.</td>
<td>Risk of incompatibility of form with the database</td>
<td>Moderate</td>
<td>No such risk</td>
</tr>
<tr>
<td>3.</td>
<td>Skills requirement for scripting to create a form consistent with tables.</td>
<td>Low</td>
<td>No skills required</td>
</tr>
<tr>
<td>4.</td>
<td>Time taken in form validation through Server</td>
<td>Moderate</td>
<td>Moderate</td>
</tr>
<tr>
<td>5.</td>
<td>Testing time to ensure successful input forms generation</td>
<td>Moderate</td>
<td>No testing required</td>
</tr>
<tr>
<td>6.</td>
<td>Delay in connecting to database before form generation</td>
<td>No delay</td>
<td>Small delay</td>
</tr>
</tbody>
</table>

2.7 FORM TESTING

The fields in the form can be auto tested by cross-field validation testing and data type integrity testing. The rule sets and test cases already developed and available can used for data errors. Regarding the rendering of the form the initial thing to verify if the form type compiles. This includes basic class inheritance, the build form function and options resolution. This should be the first test to write. This test checks that none of data transformers used by the form failed. Next, verify the submission and mapping of the form. Finally check the view of the form.
2.8 CONCLUSION

Study on automated code and generic form generation is done. The results from the analysis clearly show that, the generic code can be easily generated from the design and it is very vital in case of software engineering and computing. Auto generation of the forms is a milestone in automation. The delay caused in generation of forms by connecting to the server’s database and processing its attributes, has been traded off with the reliability and speedier implementation of the overall system. As observed through the quantitative models used for analysis, overall the algorithm proves to be more efficient than the conventional methods of form generation. It is much more reliable, reusable, generic and suitable for web services generating various forms and maintaining a large amount of client based database. However, for web services, which doesn’t require varied databases and have little interactions conventional methods are rather more appropriate. The proposed algorithm presented a generic and generative solution for form generation using the database attributes. This method is generic for any form of database and doesn’t require knowledge of scripting for creating input forms. As interactive web pages are on the rise, the proposed method could reduce significant time lost in input from creation, validation and testing.
CHAPTER 3

AUTOMATIC CODE GENERATION FOR RECURRING CODE PATTERNS IN WEB BASED APPLICATIONS AND INCREASING EFFICIENCY OF DATA ACCESS CODE

3.1 INTRODUCTION

Today, a lot of web applications and web sites are data driven. These web applications have all the static and dynamic data stored in relational databases. The aim of this chapter is to generate automatic code for data access located in relational databases in minimum time.

The code generator made by Goodarzi (2009), which is taken the basis of this study was implemented in c# on the .Net platform. It generates SQL code for the database-level part of the code, and c# or VB.Net for the application level code. There are two main approaches to code generation, often referred to as passive and active. The passive approach implies generating code only once (or re-generating it each time a modification is required). The active approach includes the option to automatically update previously generated and manually edited code (Glass 1996; Jacob et al 2006; Jensen et al 2004). The code generator used is the combination of both the approaches.
3.2 PREVIOUS WORK

The database-level code is generated using both approaches. The stored procedures follow the pattern similar to the application-level code; there are automatically generated procedures, which are re-generated each time the generator executes, and there are custom procedures, which are not affected by the generator. However, the tables and their structure are automatically updated. The code generator accepts input a file with the description of the application and processes it in the following steps:

1. A *Parser* object is responsible for parsing the input and generating a parse tree. The parser is also responsible for validating the syntax and structural integrity of the schema in the input file. The objects constituting the application’s abstract syntax contain detailed validation rules for each part of the application, such as checking that field lengths do not exceed their maximum values, that the data types are database-compatible, etc.

2. A *SchemaValidator* object is responsible for checking the application schema as a whole, which guards against duplicate class names, duplicate primary keys.

3. A *SchemaDatabaseLoader* object creates a *Database* object based on the schema file – which is an abstract model of the database part of the application.

4. An *SqlDatabaseLoader* connects to the application’s database and does the same based on the schema retrieved from the database. The two abstract databases are compared by a *DatabaseComparer* object, which insures that the two schemas are compatible (for example, the data type of an existing field cannot be changed to an incompatible data type: a string cannot be converted to an integer, for that might result in loss of data). The *DatabaseComparer* object exposes several collections,
including tables to create, tables to delete, tables to modify, constraints to create, etc., which are then accessed by objects responsible for generating the actual code.

5. A `DatabaseHelper` object takes the `DatabaseComparer` as input, generates all the database-level code, connects to the database and updates it based on the data provided by the `DatabaseComparer`.

6. An `ApplicationLoader` object takes the parse tree as input and creates an abstract syntax tree, which is an abstract model of the application. This object is passed on to several objects, which generate the actual code. The data intensive web applications consist of three components:

   1. Data Access Layer
   2. Business Logic Layer
   3. Presentation Layer

   The application logic layer is unique for an application and does not contain recurring code patterns. Thus it is not suitable for automatic code generation. Presentation layer is the user interface like desktop application. It does not require automatic code generation.

   The data access layer requires automatic code generation for producing recurring data access code that is used by different modules of the web application. Goodarzi (2009) had made a model in XML and code generator made by him generates data access code for Microsoft .NET/SQL server platform. The code generator produces at least 50% of data access code based on specifications provided in the data model but only 20-35% of data access code is used by the application.
3.3 LINK RANK - THE NEW APPROACH

Relational link is a pointer from one data unit to another. There is a lot of code which is unused with respect to application because code generator generates automatic recurrent code used by all applications in a website. Thus to increase the efficiency of data access code, modifications need to be made in data retrieval algorithms. The proposal is to introduce the concept of “Link Rank” between the millions of link and data units in a huge database. It is based on probability distribution.

The data links frequently used by web application are given high ranks than the links between unused data units. For instance, if the database consist of data units A,B,C,D with B and D data links mostly used by the application as shown in Table 3.1.

Table 3.1 Link Rank

<table>
<thead>
<tr>
<th>Links</th>
<th>Number of Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>9</td>
</tr>
</tbody>
</table>

Since, the probability of access of the data links B and D is very high as compared to other data links A and C. Hence the data links B and D are given high ranks as compared to links A and C. The link rank algorithm is implemented analogous to Google Page Rank algorithm, but it does concentrate only on inbound links and not outbound links. The data links connected by higher ranks relational links are stored in separate servers with the previous servers acting as back-up store. This will allow the code
generator to produce data access code for frequently visited data units. Thus the efficiency of the code generated will increase manifold. This will also lessen the burden of functioning on populated sites.

The java code to rank the links in java platform is given below. This is modified version of the java code generated for Google page ranking by NimaGoodarzi.

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import Jama.Matrix;
public class Ranking {
    private final double DAMPING_FACTOR = 0.85;
    private List params = new ArrayList();
    public static void main(String[] args) {
        Ranking ranking = new Ranking();
        System.out.print(ranking.rank("C");
    }
    /* * * solve the equation of ax=b, which: a is the generated matrix based on * 
    * the parameter constants. x is the link ranks matrix. b is a n*1 matrix* * 
    * which all the values are equal to the damping factor. */ */
```
public double rank(String linkId) {
    generateParamList(linkId);
    Matrix a = new Matrix(generateMatrix());
    double[][] arrB = new double[params.size()][1];
    for (int i = 0; i < params.size(); i++) {
        arrB[i][0] = 1 - DAMPING_FACTOR;
    }
    Matrix b = new Matrix(arrB);
    // Solve the equation and get the link ranks
    Matrix x = a.solve(b);
    int ind = 0;
    int cnt = 0;
    for (Iterator it = params.iterator(); it.hasNext();)
    {
        String curlink = (String) it.next();
        if (curPage.equals(pageId))
        ind = cnt;
        cnt++;
    }
    return x.getArray()[ind][0];
}

private double[] generateMatrix() {
    double[][] arr = new double[params.size()][params.size()];
    for (int i = 0; i < params.size(); i++) {
        for (int j = 0; j < params.size(); j++) {
            arr[i][j] = getMultiFactor((String) params.get(i), (String) params.get(j));
        }
    } return arr;
}

/* This method generates the matrix of the linear equations. The generated matrix is a n*n matrix where n is number of the related pages. */

/* This method returns the constant of the given variable in the linear equation. */
```java
private double getMultiFactor(String sourceId, String linkId) {
    if (sourceId.equals(linkId))
        return 1;
    else {
        String[] inc = getInboundLinks(sourceId);
        for (int i = 0; i < inc.length; i++) {
            if (inc[i].equals(linkId)) { return -1;
        }
    }
}

private void generateParamList(String pageId) {
    // Add the starting page.
    if (!params.contains(pageId)) params.add(pageId);
    // Get list of the inbound pages
    String[] inc = getInboundLinks(pageId);
    // Add the inbound links to the params list and do same for inbound links
    for (int i = 0; i < inc.length; i++) {
        if (!params.contains(inc[i]))
            generateParamList(inc[i]);
    }
}

/* This method returns list of the related pages. This list is also the parameters in the linear equation*/
private String[] getInboundLinks(String pageId) {
    // This simulates a simple page collection
    Map map = new HashMap();
    map.put("A", new String[] { "C" });
    map.put("B", new String[] { "A" });
    map.put("C", new String[] { "A", "B" }); return (String[]) map.get(pageId);
}
```

/* This method returns list of the related pages. This list is also the parameters in the linear equation*/
private void generateParamList(String pageId) {
    // Add the starting page.
    if (!params.contains(pageId)) params.add(pageId);
    // Get list of the inbound pages
    String[] inc = getInboundLinks(pageId);
    // Add the inbound links to the params list and do same for inbound links
    for (int i = 0; i < inc.length; i++) {
        if (!params.contains(inc[i]))
            generateParamList(inc[i]);
    }
}

/* This simulates a simple page collection
Map map = new HashMap();
map.put("A", new String[] { "C" });
map.put("B", new String[] { "A" });
map.put("C", new String[] { "A", "B" }); return (String[]) map.get(pageId);
}*/

/* Return list of the inbound links to a given page.*/
private String[] getInboundLinks(String pageId) {
```
3.4 CONCLUSION

The performance of this system needs additional but expensive servers to store the backup of the data which is not retrieved much. The boost in the price of the latest servers / cloud will not be as important when compared to the competence while using the web sites and applications. Moreover, the user interface will also be better.