8-2017

Database Auto Awesome: Enhancing Database-Centric Web Applications through Informed Code Generation

Jonathan Adams
Utah State University

DATABASE AUTO AWESOME: ENHANCING DATABASE-CENTRIC WEB APPLICATIONS THROUGH INFORMED CODE GENERATION

by

Jonathan Adams

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Curtis Dyreson, Ph.D. Major Professor	Minghui Jiang, Ph.D. Committee Member
Kyumin Lee, Ph.D. Committee Member	Mark R. McLellan, Ph.D. Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2017

ABSTRACT

Database Auto Awesome: Enhancing Database-Centric Web Applications through Informed Code Generation

by

Jonathan Adams, Master of Science
Utah State University, 2017

Major Professor: Curtis Dyreson, Ph.D.
Department: Computer Science

Database Auto Awesome is an approach to enhancing in-situ, web-based, relational database applications through informed code generation. It is inspired by Google's Auto Awesome tool, which provides automatic enhancements for photos. Database Auto Awesome aims to automatically or semi-automatically improve an application by generating an enhanced set of forms and scripts to query and manage data, interfaces to other tools, content management system integration, and database mediation and migration scripts.

This thesis focuses on creating an enhanced set of forms and scripts through informed code generation. Database Auto Awesome allows application administrators who are not experts in software development or computer science to enhance an application through creating new or modified forms, form validation scripts, and database interface functions. It becomes informed by gathering data from the existing application, including forms and their processing functions, database tables and columns, and database column data types and content restrictions. Using this information, several valuable enhancements can be made to the application with very little input from the administrator. First, forms can be enriched with new form validation scripts, complete with user friendly error messaging. These validation scripts are generated using the data types of the data targeted by the form fields and unique or foreign key constraints that exist on the same targeted data. Second, new database manipulation scripts can be generated targeting specific tasks and data within the database. Third, new forms can be generated, consisting of the user interface, validation scripts, and database manipulation scripts, making a completely functioning form to expand or improve the functionality of the application.

These enhancements are directed by an administrator specifying what they would like to have generated, in terms of functionality. The other requisite information to build these functioning code blocks is based entirely on the information gathered by the tool and does not require the input of a software developer. Using these techniques, Database Auto Awesome provides a viable solution for semi-automatically generating enhancements to an existing web application.

(57 pages)

PUBLIC ABSTRACT

Database Auto Awesome: Enhancing Database-Centric Web Applications through Informed Code Generation

Jonathan Adams

Database Auto Awesome is an approach to enhancing web applications comprised of forms used to interact with stored information. It was inspired by Google's Auto Awesome tool, which provides automatic enhancements for photos. Database Auto Awesome aims to automatically or semi-automatically provide improvements to an application by expanding the functionality of the application and improving the existing code.

This thesis describes a tool that gathers information from the application and provides details on how the parts of the application work together. This information provides the details necessary to generate new portions of an application.

These enhancements are directed by the web application administrator through specifying what they would like to have generated, in terms of functionality. Once the administrator has provided this direction, the new application code is generated and put in updated or new files. Using this approach, Database Auto Awesome provides a viable solution for semi-automatically generating enhancements to an existing web application.

ACKNOWLEDGMENTS

I would like to thank Dr. Curtis Dyreson for providing the inspiration for this project and for countless amounts of guidance. Additionally, I would like to thank Dr. Kyumin Lee, Dr. Minghui Jiang, and Dr. Curtis Dyreson for making up my committee and evaluating this work.

I am very grateful to my wife, son, and daughter for putting up with not seeing me for days at a time while I worked to produce this.

Jonathan Adams

CONTENTS

Page

ABSTRACT iii

PUBLIC ABSTRACT v

ACKNOWLEDGMENTS vi

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1
  1.1 Database Auto Awesome 1

2 RELATED WORK 6
  2.1 Taxonomy 6
  2.2 Database Usability 6
  2.3 Application Usability 11

3 DATABASE AUTO AWESOME 16
  3.1 DBApp Enhancement 16
  3.2 DBAA Tool 17
  3.3 Preprocessing 17
  3.4 Generate Database Connectors 18
  3.5 Generate Form Validation 27
  3.6 Identify Forms With Invalid Database Field References 31
  3.7 Generate New Forms 33

4 RESULTS 41
  4.1 Reduced Knowledge Required 41
  4.2 Reduced Development Time 42
  4.3 Improved Accessibility 43
  4.4 Challenges 43
  4.5 Future Work 44

5 CONCLUSION 46

REFERENCES 47

LIST OF FIGURES

Figure Page

1.1 A graph of class relationships. 2

1.2 Symbiota's cyclomatic complexity. 3

1.3 A radiograph summarizing Symbiota's software metrics. 4

2.1 Taxonomy of the surveyed papers. 7

3.1 Information gathered during preprocessing. 19

3.2 Class diagram of the analyzed code and database information. 20

3.3 Database connection settings for Symbiota. 21

3.4 Generated blocks of a DBAA insert connector. 22

3.5 Processing order of data gathering and modification for delete connector. . 25

3.6 Options for configuring a delete connector and the code generated. 26

3.7 Foreign key constraints that must be handled manually by the delete connector. 27

3.8 Update form validation UI. 30

3.9 Potentially broken forms shown in DBAA UI. 32

3.10 Sections of a DBAA generated form. 33

3.11 Form type, table, and column selection. 34

3.12 Example form preview during form layout step. 36

3.13 Options for configuring a select type form. 37

3.14 New form validation options. 38

3.15 Generated form code and file save locations. 39

3.16 Data sources that inform each portion of code generation. 40

4.1 Knowledge required to perform DBApp modifications in various areas with or without DBAA. 42

CHAPTER 1

INTRODUCTION

1.1 Database Auto Awesome

At the heart of many scientific pursuits is the gathering and cataloging of data. Modern technologies allow this data to be stored in online databases, allowing for greater ease of access and collaboration. Examples of these applications are the in-situ, web-based, relational database applications for bioinformatics and biodiversity data. We will call such an application a *DBApp*. A DBApp is a collection of web forms and processing scripts to manage and query data stored in a relational database; typically, in a three-tier architecture (client, web server, and database server). Example DBApps include Symbiota, Specify, and Genbank.

As a canonical DBApp, consider the Symbiota [1] project. Symbiota is a software platform for creating voucher-based biodiversity information portals and communities. The biodiversity data is stored in a MySQL database with 146 tables. Symbiota is written in HTML, PHP, and JavaScript, and consists of over 150,000 lines of code spanning over 500 files. That includes approximately 400 PHP classes and over 3,500 PHP methods (functions or procedures). The PHP classes are interrelated as depicted in Figure 1.1. Each class is listed around the outside of the circle. An edge connects a class that calls (invokes a method) in another class.

Software metrics can be used to estimate the complexity of software and the difficulty of extending it, maintaining it, and fixing bugs [2]. The software metrics for Symbiota were computed using PhpMetrics [3]. Symbiota scores poorly on several metrics. For example, Figure 1.2 depicts its cyclomatic complexity (number of paths through the code). Large circles are classes with high complexity, which impairs maintainability and code correctness.

A summary evaluation of Symbiota's software metrics relative to a representative average of other PHP projects evaluated by PhpMetrics is shown in Figure 1.3 where the

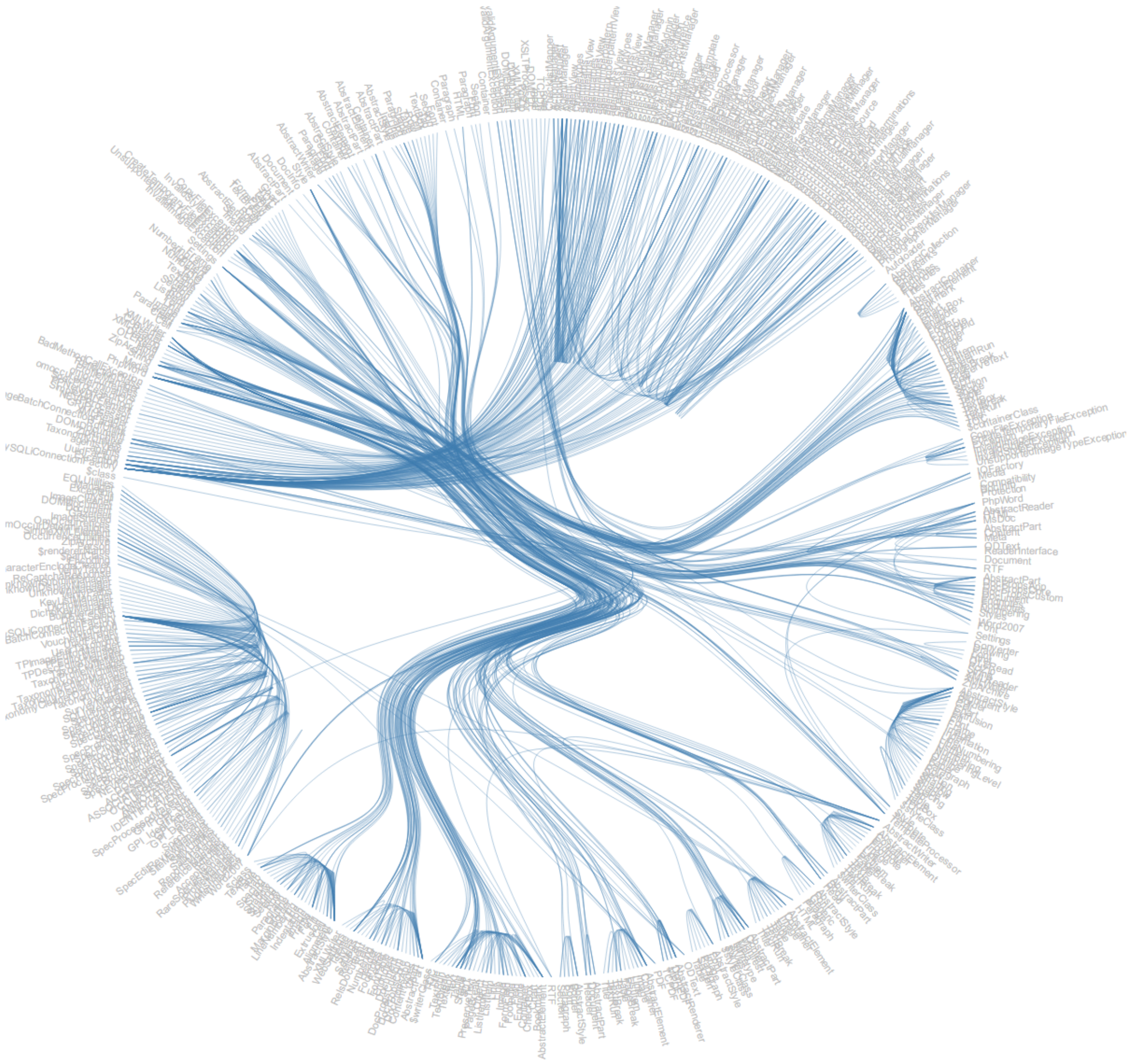


Figure 1.1: A graph of class relationships. Classes are listed around the circle. An edge between a pair of classes indicates that a method in one class calls a method in another.

metrics related to a topic (*e.g.,* development) are plotted on a radiograph. This radiograph shows that Symbiota scores low in all areas, including maintainability and accessibility for new developers.

One reason why DBApps may have poor software metrics is that their functionality is highly integrated. In Symbiota, a single PHP file may combine code for the user interface, database queries and updates, and other processing functionality. To change the user-interface, for example to create a responsive design adapted to a mobile platform or in-

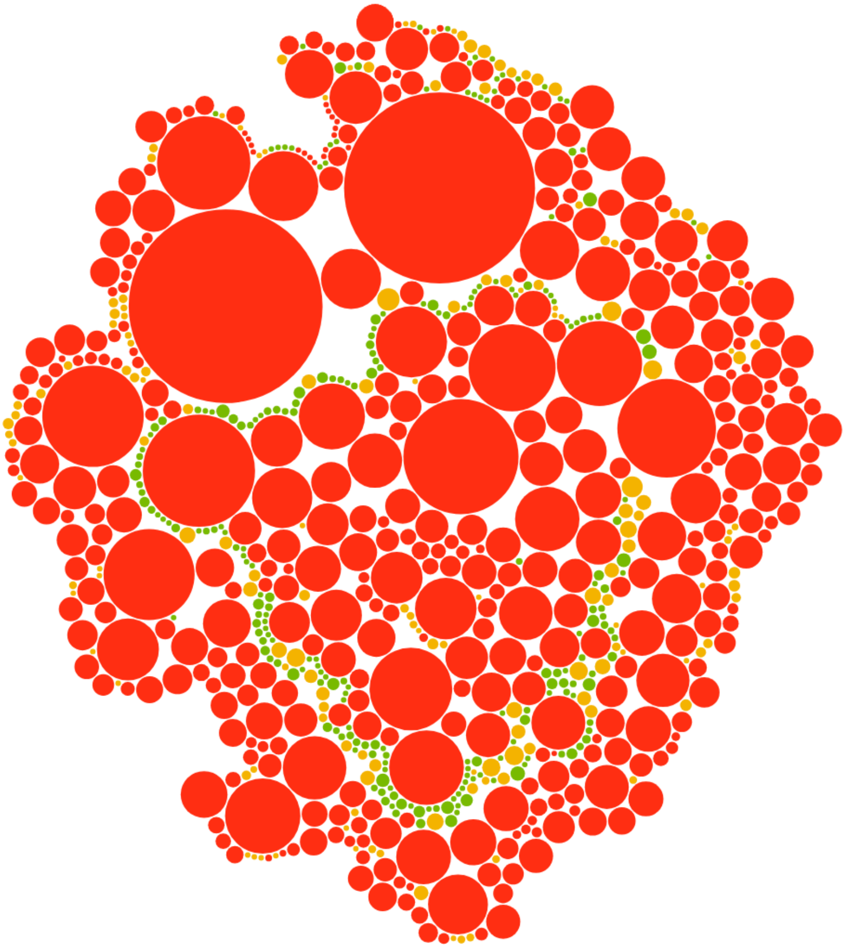
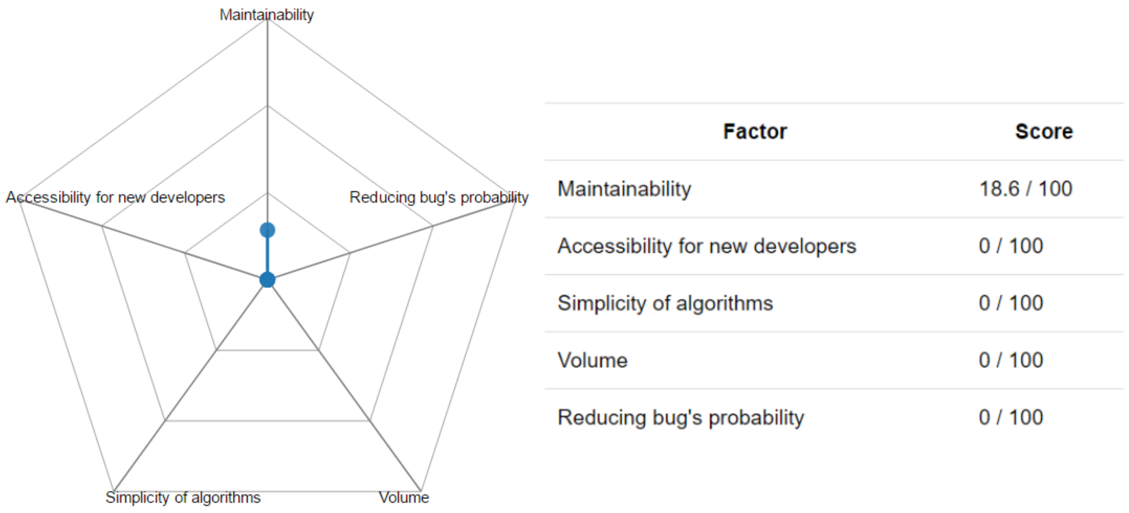


Figure 1.2: Symbiota's cyclomatic complexity. Each circle is a file whose size represents its cyclomatic complexity.

ternationalize the interface (*e.g.,* change the language in the forms from English to Arabic), reprogramming of hundreds of files might be needed.

A second reason is that DBApps often have long development lifetimes and grow over time from a small core with just a few database tables, forms, and scripts, into projects that have dozens of tables and hundreds of scripts. As new functionality is needed, it is added to the existing application, increasing the complexity of the code.

A DBApp is usually set up and run by people or organizations who did not develop or maintain the software. We will refer to these people as DBApp administrators, or simply administrators. DBApp administrators may wish to modify, improve, or extend some as-



Factor	Score
Maintainability	18.6 / 100
Accessibility for new developers	0 / 100
Simplicity of algorithms	0 / 100
Volume	0 / 100
Reducing bug's probability	0 / 100

Figure 1.3: A radiograph summarizing Symbiota's software metrics.

pects of the software to better suit their needs. In many cases the people most interested in running a DBApp are experts in their own fields, rather than in computer science or software development.

This difference in expertise, compounded by the low maintainability and accessibility for new developers, creates the need for software tools that will enable the administrators to make use of the database applications, without having to become experts in software development as well.

It is usually too costly to redesign and reprogram DBApps. What we need are cheap, effective ways to transform existing applications to suit the current needs of users and administrators. Programming efforts to maintain and extend DBApps will remain important. The success of many open source software projects shows the power and cost effectiveness of cultivating contributors to an open source DBApp project. We propose adapting Google's Auto Awesome philosophy to applications like DBApps.

Auto Awesome is a Google application that automatically enhances photos uploaded to Google+ Photos. They are enhanced by adding special effects and by combining photos to create animations and panoramas. For instance, a picture of a dog in snow may be enhanced with a falling snow effect, or combined with other photos of the dog to create an animated scene. Auto awesome also creates slide shows with music and can geo-locate (non-gps tagged) photos via photo matching. These enhancements are completely automatic. The only human input is to decide whether to keep or discard the enhanced photos. Not every photo is enhanced, only some within a collection are chosen for enhancement.

Approaches that automatically improve a project can help to lower the cost of improving DBApps. These tools would include a range of software to perform fully automated or semi-automated modifications and extensions, code error identification, and refactoring opportunity identification. There are many areas in which an Auto Awesome approach could produce enhancements to a DBApp, such as:

1. an enhanced set of forms and scripts to query and manage the data,
2. interfaces to other tools (*e.g.,* data mining with Weka or search with Sphinx),
3. content management system integration (*e.g.,* Wordpress plugins), and
4. database mediation and migration scripts.

In all cases a system implementer would be allowed to choose which enhancements to keep and use. For this work, we focus on producing enhancements in regards to the forms and scripts for querying and managing data and on the benefits of informed code generation on the extensibility and maintainability of DBApps. Since our tool takes inspiration from Google's Auto Awesome, we call it Database Auto Awesome, or DBAA.

CHAPTER 2
RELATED WORK

2.1 Taxonomy

There is work currently being done in many areas related to tools for improving the quality of existing software. The papers surveyed here all have elements related to the work of automating the improvement of software. See Figure 2.1 for the full taxonomy breakdown. Within the work of automating the improvement of software, there are two areas that are focused on, database usability and application usability.

The improvement of database usability involves using and modifying both the schema and the data within the database. Usability can be improved through better methods for retrieving data, and for managing the data. The data can also be used to inform automatic improvements in other areas. Usability can also be improved through refactoring the schema to create a better one, extending it to support a wider range of data, or by migrating it to a different database management system.

To improve the usability of the application, a tool may target the overall performance of the application, or it may offer ways to modify it through its underlying code. These modifications may consist of generating new code, refactoring existing code, or offering better user interfaces.

2.2 Database Usability

The existing database data, schema, and methods for accessing data may not provide the experience the DBApp administrator desires when deploying one of these database applications. This could be due to the database not having a location to store some attributes the administrator wants to store. It could be that the database does not return all data they would expect to be returned for a query. It could be one of many other possible issues,

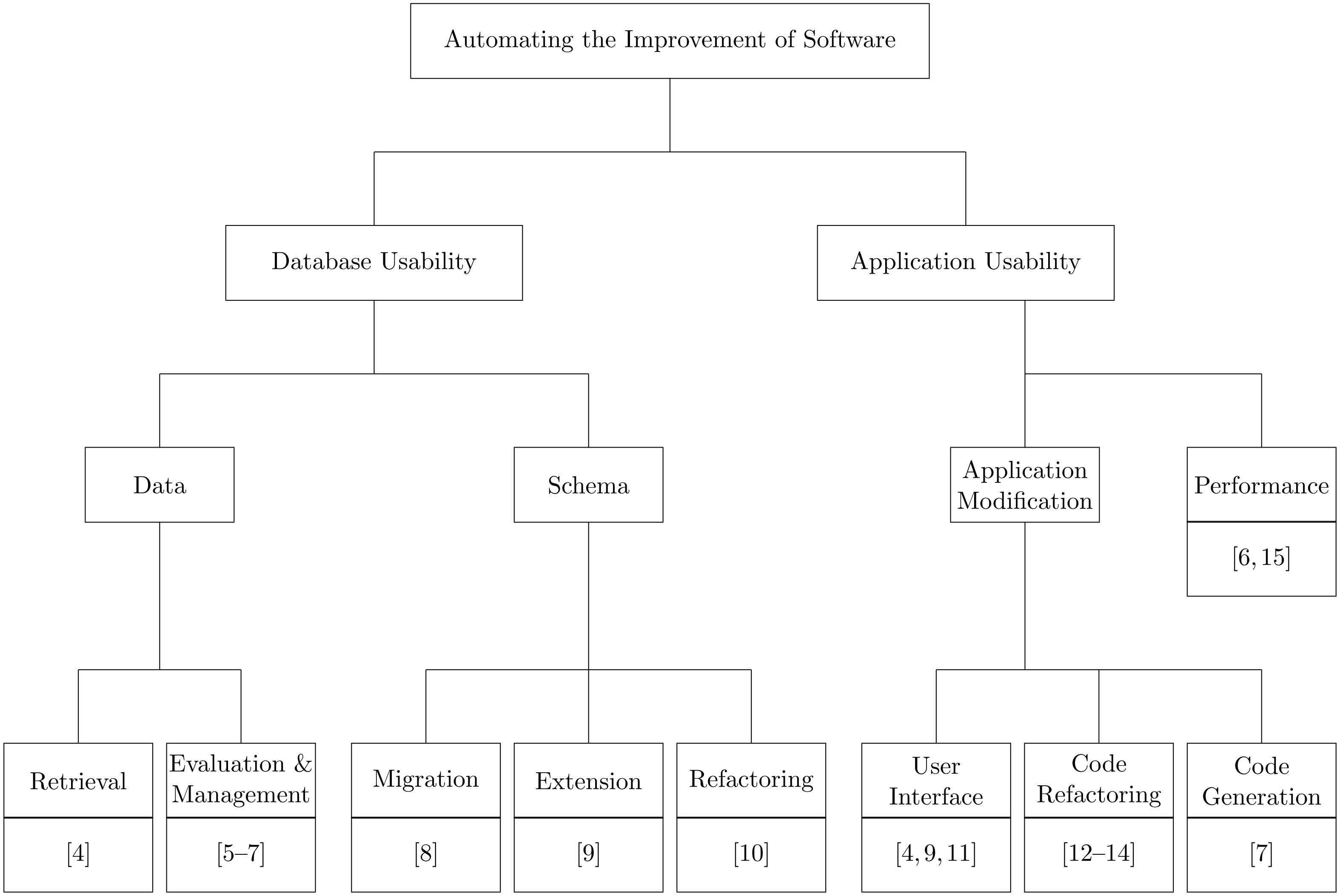


Figure 2.1: Taxonomy of the surveyed papers.

and there have been many attempts to use the underlying data and database schema to improve database applications.

2.2.1 Data

A database application's existing data can guide improvements to the application.

Retrieval

In many cases, an invalid query is not identified until the user has finished building their query through whatever user interface the application provides, and then attempts to execute that query. To determine query validity, Nandi and Jagadish [4] evaluate the schema to build a reachability index. This index is a matrix containing Boolean values relating the database elements as descendants of one another. As a query is built by the user, the attributes being searched for can be checked against this matrix to determine if the query will be valid. This approach aids in giving immediate feedback to the user, and helps them form a query that will be valid and return the correct data.

Data Evaluation and Management

Using the existing data, we can understand the nature of relationships in the data and use that information for improving the overall user experience or helping the DBApp administrator make desired changes to the system. Some databases may be populated with tuples that are incomplete because they include null values in attributes used in a query. These tuples will be left out of query results when searching on those attributes. Wolf et al. [5] propose a system, QPIAD, for predicting these missing values. This process allows tuples to be returned as possibly relevant results to these queries, even if it contains null values for attributes used in the query. This is accomplished by mining the attribute correlations, and using the existing database data to build classifiers which can predict the missing values based on other data in the tuple.

To determine possible values for missing ones in a tuple, the QPIAD system needs to understand the relationship between the database attributes. The type of relationship focused on here is functional dependencies. An example of a functional dependency would be that, given the model of a car, we should be able to determine its make. Functional dependencies are not common enough in most data sets, so they describe approximate functional dependencies, which are functional dependencies that hold true for all but a small fraction of the data. Once these have been identified, classifiers can be built to predict the probability of a given attribute value based on the other given attributes.

Similar to these approximate functional dependencies, Ilyas et al. [6] analyze the schema to find what they call soft functional dependencies. These soft functional dependencies identify the relationships between attributes where the value of a given attribute determines the dependent attribute with a high probability, not a certainty. These dependencies are scored and given a strength rating, with the highest rated dependencies stored for later use, while discarding the low rated dependencies to improve overall performance in making use of them.

Jayapandian and Jagadish [7] describe a system that reads and analyzes the design of the database schema and the data contained within, to calculate a set of metrics that they have outlined as representing the relationships within the data. These metrics include and are based on characteristics such as cardinality, convergence, participation ratio, queryability, and attribute necessity. Together, these metrics help identify which parts of the database are most likely to be queried, and when queried, which attributes are generally used together.

2.2.2 Schema

The schema describes the structure or organization of the data, which may need to have changes or improvements made to it, to fit the DBApp administrator's needs. They may need it to be converted to work with a different database management system (DBMS), extended to store new data attributes, or refactored to improve usability and performance.

Migration

Given requirements from a DBApp administrator or their institution, a DBMS other than the one in use by the application, may be needed. The schema being used will need to be converted into a schema compatible with the new DBMS. Similarly, all data will need to be imported from the existing DBMS. Fortunately, many DBMS providers have tools that will guide someone through the migration of a database from a competitor's DBMS to their own. For example, Microsoft provides the SQL Server Migration Assistant [8] for migrating from Access, DB2, MySQL, Oracle, or Sybase ASE to Microsoft's SQL Server. This is done in several steps including connecting to both the old and new DBMS, mapping the old database schema to a new schema, and then converting the data and importing into the new system.

Extension

An et al. [9] allow for a database schema to be extended automatically, as a DBApp administrator needs. This is done in response to a web form being created that needs to store or access data that is not currently found in the database schema. It determines what database fields are being used by the form and identifies form fields that relate to data not currently found in the database schema. It will then connect forms to existing database fields or extend the database as needed to accommodate the design of the forms. This will allow the administrator to approach the creation of new web forms as the most important aspect of usability, and to perform such tasks without the requisite background knowledge of how to modify a database schema.

Refactoring

A key element of these applications is the software-to-database interactions. To ensure good database usability we need to evaluate the quality of our database design. Vial [10] discusses many of the challenges of analyzing and improving a database schema, as well as techniques they found valuable in performing database refactoring. One of the key points made was the value of automating much of the work. They have created and put into use a tool, called Refactor, to analyze a relational database and to aid in common database refactoring tasks. This tool uses design guidelines built by experts, and compares those guidelines to the schema, in order to identify the proper refactoring that needs to be done. In their work, partially automated refactoring helped create consistent results even by team members who were not experts in the types of changes being made.

2.3 Application Usability

On the other side of the DBApp, we have the application that will be used to interact with the underlying database. These systems are often built from the viewpoint of a small group of people who are responsible for developing the software. People from other organizations may have needs that differ somewhat in how the application is used. Key aspects that may need changes are the user interface, the quality and structure of underlying code, and the performance of the system. These all have a significant impact on the usability of the database application.

2.3.1 Application Modification

Work to help a DBApp administrator improve the application needs to address several parts of the overall application. It must address the user interface, which usually comes in the form of a set of web forms. It must address the ability to refactor the underlying code that may not be directly observed by the user or administrator. It must also address the opportunity to generate new code to meet needs, which are not being met by the application as it was originally designed.

User Interface

In an effort to make database systems more usable, Jagadish et al. [11] have identified several usability pain points. They observe that an application must be simple enough that a novice user can effectively accomplish their tasks, while allowing expert users the flexibility to perform more advanced tasks. Part of their work was creating and testing alternatives to standard web forms. Traditional web forms can be replaced with search engine style keyword searches, natural language query processing, code based query building, and guided query building graphical user interfaces. They found that depending on the needs of the users, distinct types of interfaces had various benefits. They also found that the variations between results of the interfaces caused confusion and concerns in users that did not understand why the differences occurred.

An alternative user interface for querying a large database system is proposed by Nandi and Jagadish [4], which allows a user to be guided through building a query using a single text box. The text box dynamically displays data attributes that can be used to target data, as well as suggested values to use as matches to data for a given attribute. The system allows you to query the database based on many different criteria, without requiring unique web forms built for each type of query. This results in a very minimalistic, yet powerful interface. On the other hand, this requires the user to have at least a basic understanding of what data is found in the database and how it is organized so they can begin to type the correct keywords to find the desired search terms.

The issue of form and database usability can be approached from the other end, by starting with the design of the web forms. As mentioned in regard to schema extensions, An et al. [9] take the approach of allowing a user to build a web form or set of web forms first and then their system connects the forms to the database. Using information observed about the database schema, it determines which database fields are being used by the form and identifies the form fields that relate to data that is not currently found in the database schema. It will then connect forms to existing database fields or extend the database schema as needed to accommodate the design of the forms. This will allow the designers to approach the design of the web forms as the most important aspect of usability, and to adjust the entire database application to match those needs.

Code Refactoring

Beyond the usability of the interface there are other pain points related to a DBApp administrator modifying, extending, or customizing the application. In a traditional development procedure, the person making these types of changes must be familiar with much or all of the application and the database. The usability of a database application, from the viewpoint of one of these novice administrators, is determined by how effectively they are able to make the needed changes to the application. In the case of those with very little software development experience, even the most well structured and well written code bases will not be easily modifiable. A very important factor of database usability is having tools to aid in modifying, improving, and expanding on the code base.

In order to make these types of modifications, the first key step is identifying opportunities to refactor the code and improve the application. As Cedrim [12] discusses, the finding of refactoring opportunities is usually performed by experienced software developers who can use their intuition to spot code that needs improvement. A less experienced administrator of a database application will not have the necessary experience and knowledge to make proper informed decisions about refactoring the code. To automate the identification of such opportunities, they propose a system to make use of machine learning models that have been trained using identified and classified refactorings. Based on this past data, they hope to be able to identify and classify new opportunities that match similar code patterns.

Sharma [13] takes a different approach to identify refactoring opportunities specific to the extract-method class of refactoring. This type of refactoring involves pulling an existing section of code out into a separate method, without altering the overall behavior of the code segment. This is done to reduce complexity of segments of code. They accomplish this by generating graphs covering the way the code is executed and determining the data dependencies of each segment of code. Using this data, they can identify code that is a good opportunity to extract into a separate method, without disrupting the function of that code.

Xin et al. [14] have created a tool, called MEET DB2, that will analyze the code in order to understand what actions are performed on the database by the code. Its goal is to identify areas that will require modification if the underlying DBMS is changed to a different DBMS. It works by parsing and analyzing the application's code. It then considers the capabilities and functionality of the new DBMS while determining what functionality is used by the code. If the code uses certain functionality that is not compatible with the new DBMS, it attempts to estimate the work required to make that portion of code compatible.

It will compile potential problems and provide a report. For example, it will detail how many lines of code will need to be modified to make your application compatible, where those lines of code are found, and what problems were found in the code. This can greatly aid in identifying the challenges associated with migration.

Code Generation

One task an administrator may have is the creation of new forms used to access or modify the data. As mentioned in discussing data evaluation above, Jayapandian and Jagadish [7] proposed a solution to automatically identify the important relationships in a database. Using this information, they can propose a set of forms that will cover most queries that are likely to be performed on the data. That set of forms can then be automatically generated. Given that these forms should cover the predicted set of necessary queries, this may result in all the forms necessary to use the database. This approach has the benefit of being fully automated and requiring no understanding of the existing database schema or its data. The automatic nature of the system, however, may not produce forms that match all the specific requirements the administrator has.

2.3.2 Performance

An administrator may find that the database application they are setting up does not perform well enough for their needs. This could be performance in either speed of executing queries or storage space usage.

Zisman and Kramer [15] propose a system to improve performance in systems made up of many autonomous databases. Their system aims to accomplish this by removing the number of databases searched to discover information, as well as to remove the need for centralized systems to direct the searches, which can result in performance bottlenecks. This is accomplished by analyzing the contents of each database and building data structures that help identify the locations of various types of data. Using that, they can recursively search the various databases, and get to the requested information quicker.

Ilyas et al. [6] use the stored soft functional dependencies, discussed above in relation to data analysis, to provide statistics on the underlying data. This data is used to help query optimizers produce faster queries. This is done by helping identify the selectivity of these relationships. This automatic process of determining the relevant information, would allow an administrator to improve the performance of the queries performed by the application, without digging into all of the details of the database. Without an automated tool like this, the improvements would require the assistance of someone trained in database administration.

CHAPTER 3
DATABASE AUTO AWESOME

3.1 DBApp Enhancement

Extending the functionality or improving the quality of a DBApp requires the modification of many different components of the application. Symbiota is structured as a web front end made up of HTML and JavaScript, with PHP and a MySQL database on the back end. Extending functionality by adding a new form for accessing the data would require building a new form in HTML, creating new validation and form processing scripts in JavaScript, and creating the PHP code to interact with the database. Similarly, updating the form validation scripts to be more robust or to have better error messaging could require modifying HTML, JavaScript, and PHP code.

There are many possible reasons why someone would want to extend or modify the capabilities of a DBApp. There may be a need for users to access data that cannot be retrieved due to the lack of a form. Changes may be made to the underlying data structure that require new or updated forms. An administrator may want to build a custom set of forms that presents a small subsection of the overall DBApp to simplify the user experience.

To make these enhancements to Symbiota, for example, requires an understanding of how data is stored in a relational database and how data is modified or retrieved. Additionally, experience with writing HTML, JavaScript, and PHP is required, along with an understanding of how all three work together. This skill requirement creates a barrier to modifying the application and means that a DBApp administrator, who wishes to make changes to the application, must either become a software developer, spend money on developers to make the desired changes, or request the features from the open source community and hope someone delivers. These are not always feasible options depending on the administrator, their budget, and the nature of the desired changes.

Informed code generation is the practice of creating, extending, updating, or replacing code based on information available to the system. The requisite information can come from existing code, the data stored in the database, the structure of the database, a template of what is being created, or user input. When developing a software application, developers intuitively use much of the same information to determine how new code should be written and structured. In the case of DBAA, the goal is to mimic the work done by a software developer, while requiring little-to-no input from one.

3.2 DBAA Tool

The DBAA tool uses informed code generation to enhance the forms and database interactions of a DBApp. This enhancement comes through the generation of new form validation scripts and error messaging, the generation of new database connector scripts for accessing or modifying the stored data, the identification of existing broken forms that may be trying to access portions of the database that no longer exits, and the generation of completely new forms that are fully complete, from front end to back.

3.3 Preprocessing

In order to perform the informed code generation and enhancements, DBAA must first gather information about the application. The potentially relevant information is all gathered up front, to provide a very responsive application through the enhancement process.

The primary source of information that DBAA uses is the database schema, which contains all the details about the data being stored. For example, in Symbiota the taxa table, stores taxonomical information. The schema shows that the table contains the name and rank of each taxon. Additionally, it shows that the name of the taxon is stored as a character string and can be, at most, 250 characters long. It also states that each entry in that field must be unique. The rank is stored as a number with a max value of 65,535. It also shows foreign key constraints, that is, which database fields reference other fields, restricting data modification.

The other source of information DBAA analyzes is the application code. The code is parsed and DBAA stores key features of the code, such as where the forms are found, which script the form is submitted to for processing, where form validation scripts are stored, and what database queries are performed as a result of the form being submitted. Using this information, DBAA connects form fields to the database fields they relate to. Though there is often a one-to-one correspondence, *e.g.,* a taxa name input box is connected to a `taxa.sciname` field, the correspondence can be many-to-many.

The diagram in Figure 3.1 shows how the information from the application is broken down and analyzed. The code files and database schema are processed and key elements are identified, as shown in the diagram.

As an example of how the data can be stored, a class diagram from DBAA is shown in Figure 3.2. All data is collected in the *DBAppData* class with the database schema and code information broken into their respective components.

To process this information, the user needs to provide connection information for how to access the database, and the location of the DBApp code files as shown in Figure 3.3. This information is stored as an encrypted user setting for future uses of the application. Once DBAA has this information, it can process the DBApp to gather all relevant data.

3.4 Generate Database Connectors

A DBApp's usefulness centers around being able to access or update data. To make beneficial enhancements to the application, DBAA needs to be capable of generating code to allow for these data interactions. To generate any interaction with the database, the tool needs to know how the DBApp communicates with the database. In DBAA, the configuration menu has a section where the user can provide the necessary information.

The tool allows the user to provide a list of files that must be included in the PHP database connector to function properly. It asks for the code required to build the database connection object, including separate configurations for read only and write access type connections. The settings needed for Symbiota are given as an example in Figure 3.3.

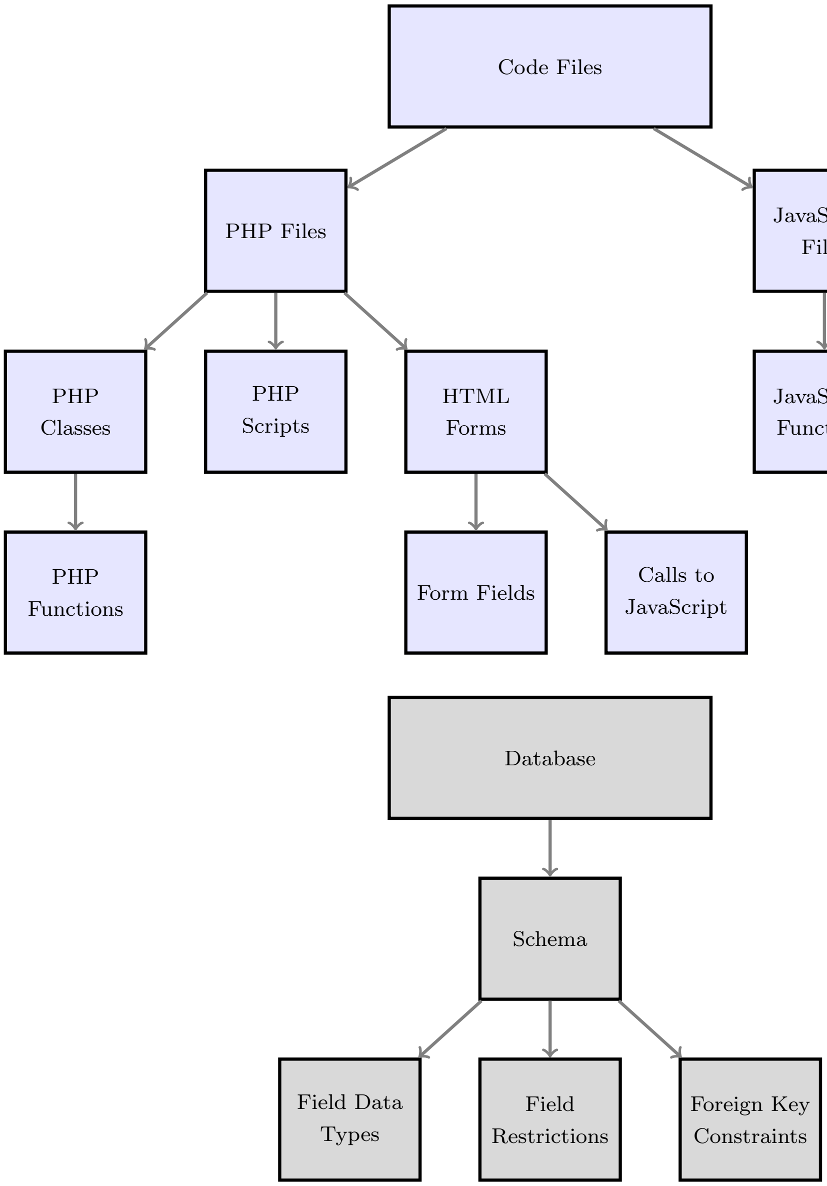


Figure 3.1: Information gathered during preprocessing.

Symbiota uses the MySQLi class to communicate with the MySQL database. It also uses a connection factory class to generate or retrieve the proper connection object.

The connectors are generated from a template set of building blocks. An example of these blocks is shown in the context of a generated insert connector in Figure 3.4. First the information provided in the configuration menu is used to include the requisite files and to create the connection object. From there the SQL statement or statements are built for the

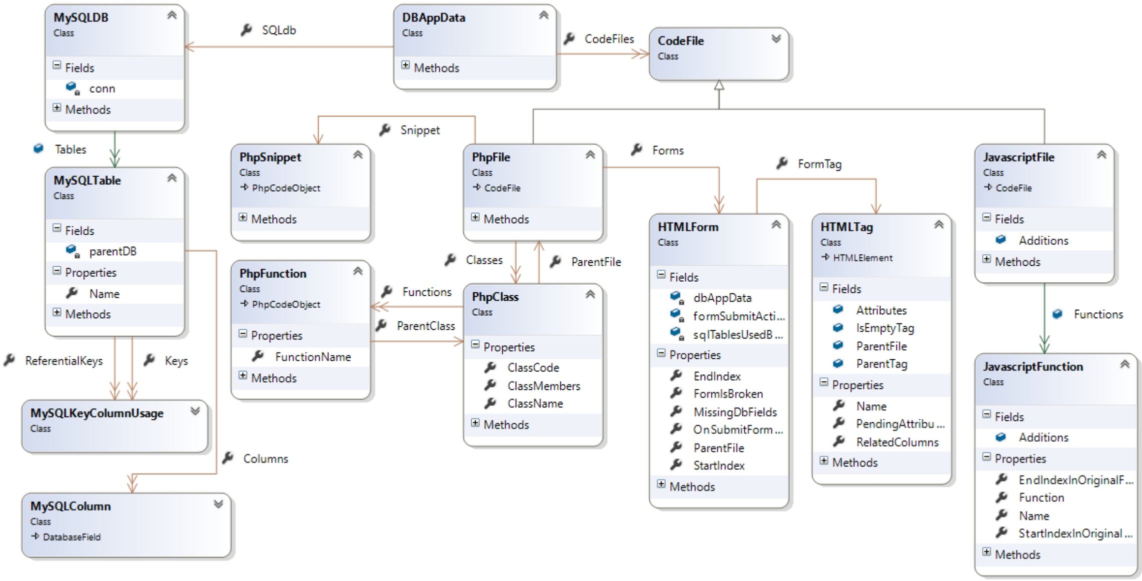


Figure 3.2: Class diagram of the analyzed code and database information.

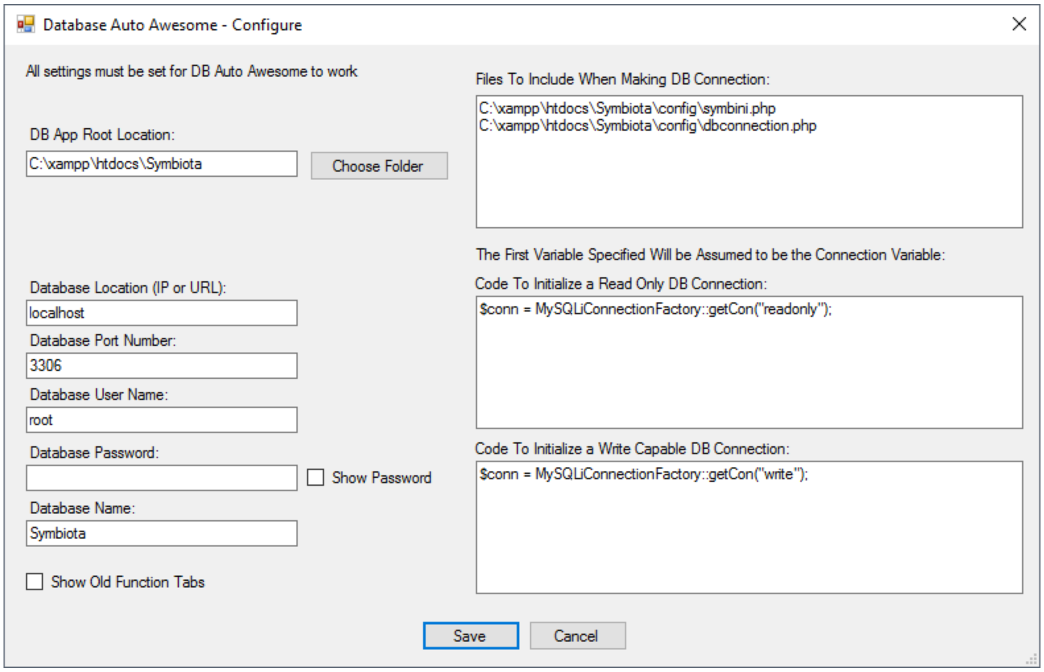
tables being used. Then the query processing code is generated for binding the submitted values, executing the SQL statement, and handling the results.

To properly generate the value binding code, DBAA needs to know the data type of each column the query interacts with. When the form input data is bound to the SQL statement, a data type needs to be specified for each item. The data types needed for the binding function of the MySQLi class do not match the data types of the database. To resolve the difference, DBAA simply retrieves the data type for each column from the schema, then uses that type to look up the matching type to use when binding the values.

There are four classes of connectors that can be generated by the tool.

1. Insert data into a table
2. Check if specific data values currently exist in a table
3. Retrieve data from one or more tables
4. Delete data from a table

The simplest interaction type is inserting data into the database. To build this type of connector, the tool needs to know which database table and which columns in that table



Database Auto Awesome - Configure

All settings must be set for DB Auto Awesome to work

DB App Root Location: C:\xampp\htdocs\Symbiota

Database Location (IP or URL): localhost

Database Port Number: 3306

Database User Name: root

Database Password:

☐ Show Password

Database Name: Symbiota

☐ Show Old Function Tabs

Files To Include When Making DB Connection:

```
C:\xampp\htdocs\Symbiota\config\symbini.php
C:\xampp\htdocs\Symbiota\config\dbconnection.php
```

The First Variable Specified Will be Assumed to be the Connection Variable:

Code To Initialize a Read Only DB Connection:

```
$conn = MySQLiConnectionFactory::getCon("readonly");
```

Code To Initialize a Write Capable DB Connection:

```
$conn = MySQLiConnectionFactory::getCon("write");
```

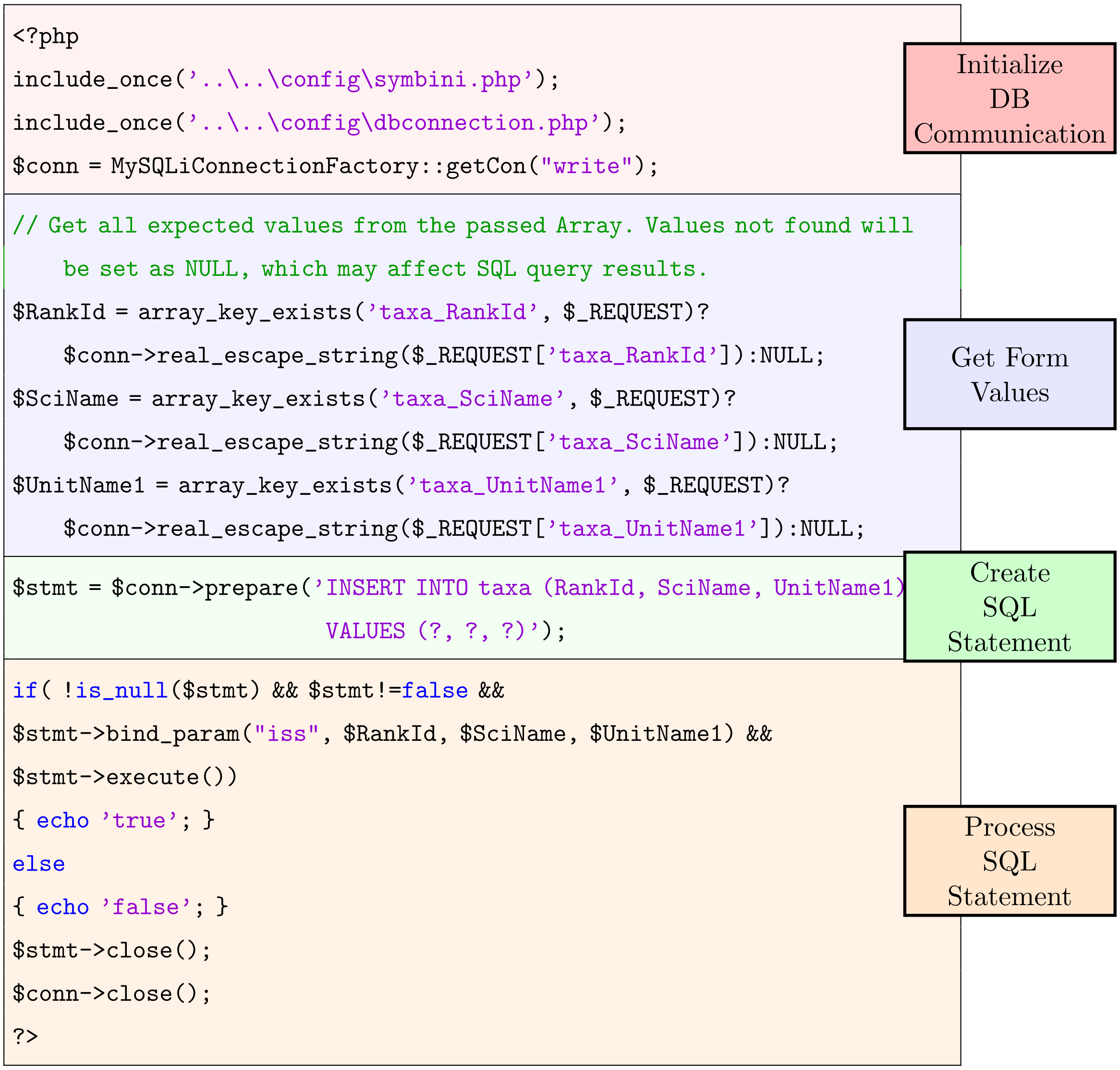
Figure 3.3: Database connection settings for Symbiota.

the data goes in. The connector will take the appropriate values, attempt to insert them into the database, and then return a Boolean to indicate whether it succeeded.

The most common interactions found in Symbiota are the two data retrieval types. The interaction to check if data values currently exist in the table is typically used to check if data being inserted into the table will meet constraints on the values, like uniqueness or a foreign key constraint. The information required to generate this type of connector is the table and columns to check against and what type of value comparison to do. Typically, this type of interaction is based on the equality operator, to ensure that the submitted values exactly match existing values. However, if the need arises, any standard SQL comparator can be used. The connector will return true or false to indicate whether the provided values match any existing values in the database.

The other data retrieval operation is used to retrieve a set of data matching a set of input values. This connector is a bit more complicated to generate, depending on the desired use, than the previous two. If a single table is desired, the tool needs to know the table to access, the columns to return values from, the columns to compare against, and the comparator type to use in each comparison. Using this, the tool can generate the necessary SQL SELECT statement to fetch the matching results, along with the necessary PHP to execute that statement and process the results.

If values from more than one table are needed, more information is required. The tool needs all items listed in the previous paragraph, but for each table accessed. It also needs the description of how to connect the tables. Currently, the DBAA tool only supports inner joins. For each join, the tool needs to know which two tables to perform the join on, and



```
<?php
include_once('..\..\config\symbini.php');
include_once('..\..\config\dbconnection.php');
$conn = MySQLiConnectionFactory::getCon("write");
// Get all expected values from the passed Array. Values not found will
    be set as NULL, which may affect SQL query results.
$RankId = array_key_exists('taxa_RankId', $_REQUEST)?
    $conn->real_escape_string($_REQUEST['taxa_RankId']):NULL;
$SciName = array_key_exists('taxa_SciName', $_REQUEST)?
    $conn->real_escape_string($_REQUEST['taxa_SciName']):NULL;
$UnitName1 = array_key_exists('taxa_UnitName1', $_REQUEST)?
    $conn->real_escape_string($_REQUEST['taxa_UnitName1']):NULL;
$stmt = $conn->prepare('INSERT INTO taxa (RankId, SciName, UnitName1)
                        VALUES (?, ?, ?)');
if( !is_null($stmt) && $stmt!=false &&
$stmt->bind_param("iss", $RankId, $SciName, $UnitName1) &&
$stmt->execute())
{ echo 'true'; }
else
{ echo 'false'; }
$stmt->close();
$conn->close();
?>
```

Figure 3.4: Generated blocks of a DBAA insert connector.

which columns to join on. The complete set of tables must form a single chain of joined tables. Provided with this extra information, the tool can generate a more complex SQL SELECT statement to fetch the matching results from the combined columns of all joined tables. A successful query will return the fetched results as JavaScript Object Notation (JSON) data, while a failed query will return an error message.

The most complicated connector to generate is the delete data from a table connector. Performing a delete operation on a single MySQL table is simple, but a table does not exist in a vacuum. A table may be referenced by other tables through foreign key constraints. To ensure the integrity of the data, a delete operation cannot be performed on any data that is currently referenced by a matching value in another table. In a sense, these restrictions pin the data so it cannot be deleted or modified, until all references are modified to no longer refer to those values. If DBAA attempts to perform the delete operation without resolving those constraints, the database will return an error and no change will be made.

Some of these constraints will be configured with an automatic response to an attempt to change the referenced values. The responses that will automatically free up the data are the *cascade delete*, *cascade null*, and *set null* responses. Cascade delete will delete all rows containing values that reference another table's value that will be deleted. Similarly, cascade null will set to null any value that references another table's value that will be set to null. Set null will set the referencing value to null if the referenced value is deleted or set to null.

If all foreign key constraints that reference the target table have one of these options set, the connector does not need to manually handle any other table and can simply perform the deletion. This is not always the case, however. In all other cases, the tables must be handled manually.

To determine what must be handled manually, DBAA refers to the database schema information it gathered. That information contains every foreign key constraint. From this information, DBAA builds a graph with tables as the nodes, or vertices, and foreign key constraints as the paths, or edges. The graph is a directed graph to preserve the direction of the constraints. The graph typically takes the form of a tree with all edges pointing from the leaf nodes upward, toward the target table, but the graph could have cycles.

The graph is built starting from the table the user wishes to perform a deletion on. Then, the tool traces each foreign key constraint that may be triggered by the deletion back to the referencing tables. Then, from those tables it traces all foreign key constraints that may be triggered. It continues in that manner until completing a graph of all possibly affected tables.

Traversing this graph, DBAA determines which constraints will be resolved automatically, and which must be resolved manually. This is based on what type of operation triggered the constraint, the field characteristics, and the constraints action for that type of event. If the source of the constraint is not a nullable field, then it needs to delete the full conflicting tuple. If the source of the constraint is nullable, then it defaults to just setting that attribute to null. The user can override that option to always delete the tuples, if desired.

Then, using this graph DBAA generates a series of database operations. First, it finds the tuples that will be deleted by the initial delete operation. Then, it finds any tuples that will be modified in other tables as a result and will trigger other manually handled foreign key constraints. For each table that has referencing tables, it gathers all tuples that will be affected.

Using the lists of affected tuples, DBAA can start at the bottom, performing the deletes and updates accordingly and moving back upward, until it performs the users desired delete action as the final operation. This way all foreign key conflicts will be resolved without losing referential integrity.

A simplified example of the procedure for processing the delete operation is shown in Figure 3.5. Here, the connector starts at the target table, the one the delete operation will be performed on, and finds the tuples that will be affected by the delete operation. Then it goes down one step, to the middle layer, and uses that tuple data to determine which tuples will need to be modified in the middle layer. Using this information, it can go to the

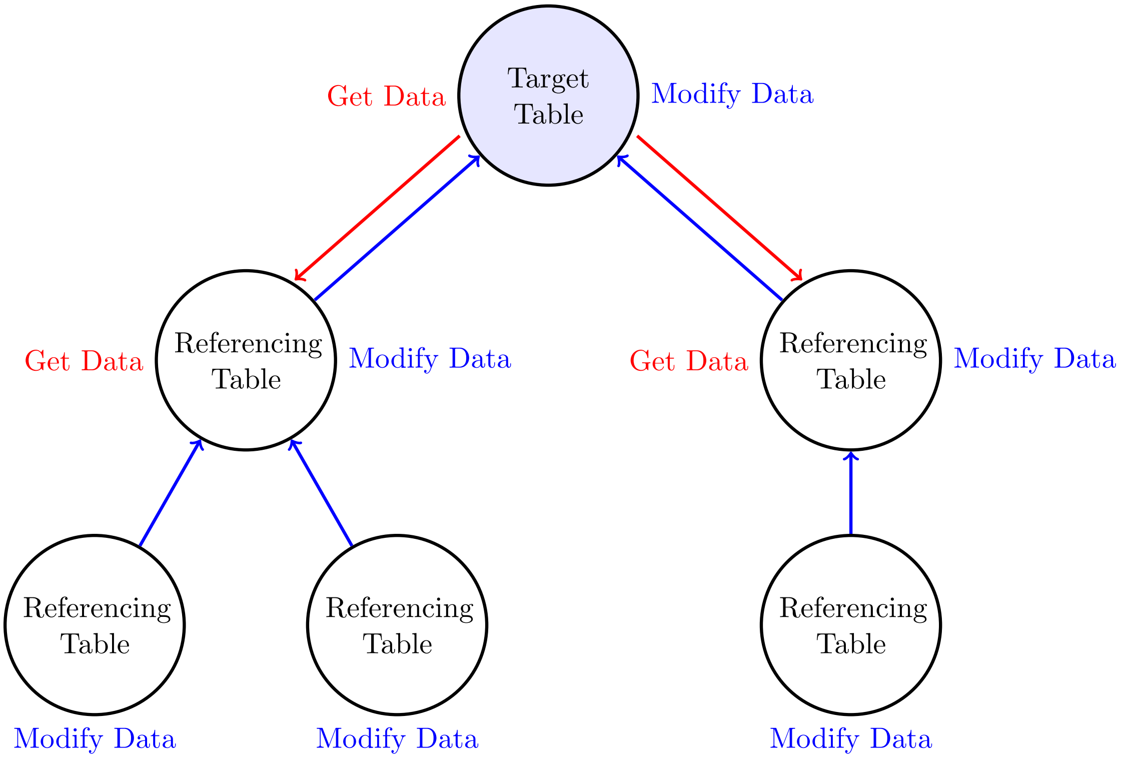


Figure 3.5: Processing order of data gathering and modification for delete connector.

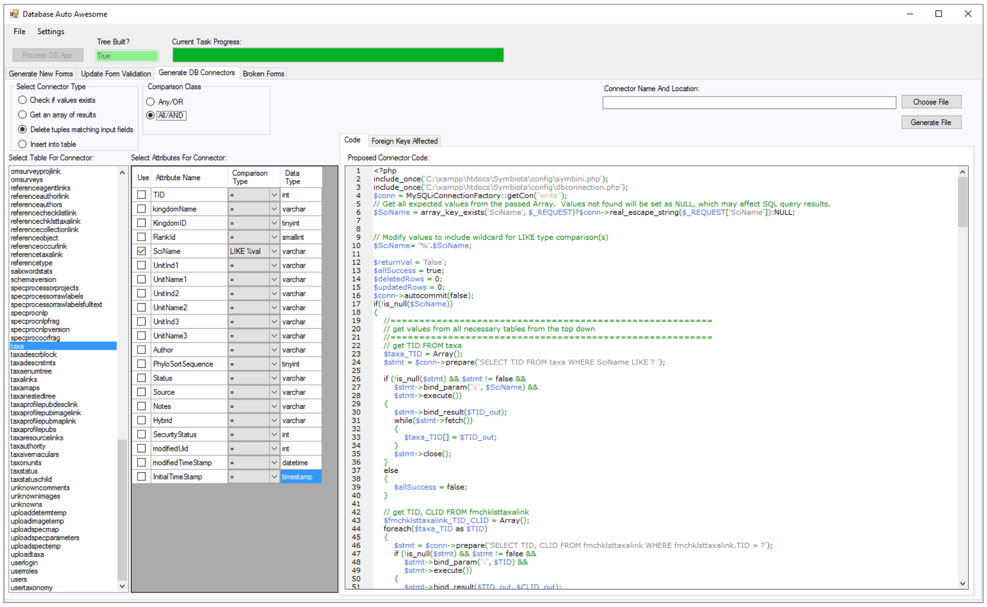
bottom layer. It doesn't need to store data for this layer, but it will use the tuples from the middle layer to modify (delete or set null) the data in the bottom layer tables.

Then, it goes back up and modifies the data in the middle layer that it previously found. It can now modify or remove it because the bottom layer has been modified and no longer points to that data. Then, it can move up to the original target. At this point the lower tables will no longer be referencing the tuples it originally intended to delete, so it can perform that deletion.

Alone, these connectors will not be of much use to an administrator that is looking to enhance the application, but they will be used in other portions of the DBAA tool. For developers that are manually making enhancements, generating these connectors would save them the work of manually writing those connector scripts. DBAA provides an interface for generating these standalone connectors. The user can select the type of connector, set the applicable options, and the tool will generate a new file containing the connector, ready to use.

As an example from the DBAA implementation, Figure 3.6 shows an interface for generating the connectors. In this case, a delete connector is being generated. At the top left, the user can select the type of connector to create and the comparison class to use. In the left column, the user can select the database table to delete from. The middle column contains a list of all columns in that table. Here, the user selects which columns should be used in the operation to determine which rows to delete. For each column, the user can choose the type of comparator to use. The default comparator is equality, but the user can select from many others including less than, greater than, or like.

The right column contains the generated code, which is constantly updated to match the settings the user has selected in the other columns. This column also has a tab which shows a list of all foreign key constraints which may be triggered by the delete operation. This is shown in Figure 3.7. All of these settings are configured to defaults that will function with no input from the user. If the user wishes to change the behavior for reacting to a foreign key constraint, they can do so here.



Tree Built? True

Current Task Progress:

Select Connector Type

○ Check if values exists
○ Get an array of results
● Delete tuples matching input fields
○ Insert into table

Comparison Class

○ Any/OR
● All/AND

Connector Name And Location:

Select Table For Connector:

omsurveyprojlink
omsurveys
referenceagentlinks
referenceauthorlink
referenceauthors
referencechecklistlink
referencechklsttaxalink
referencecollectionlink
referenceobject
referenceoccurlink
referencetaxalink
referencetype
salixwordstats
schemaversion
specprocessorprojects
specprocessorrawlabels
specprocessorrawlabelsfulltext
specprocnlp
specprocnlpfrag
specprocnlpversion
specprococrfrag
taxa
taxadescrblock
taxadescrstmts
taxaenumtree
taxalinks
taxamaps
taxanestedtree
taxaprofilepubdesclink
taxaprofilepubimagelink
taxaprofilepubmaplink
taxaprofilepubs
taxaresourcelinks
taxauthority
taxavernaculars
taxonunits
taxstatus
taxstatuschild
unknowncomments
unknownimages
unknowns
uploaddetermtemp
uploadimagetemp
uploadspecmap
uploadspecparameters
uploadspectemp
uploadtaxa
userlogin
userroles
users
usertaxonomy

Select Attributes For Connector:

Use	Attribute Name	Comparison Type	Data Type
□	TID	=	int
□	kingdomName	=	varchar
□	KingdomID	=	tinyint
□	RankId	=	smallint
☑	SciName	LIKE %val	varchar
□	UnitInd1	=	varchar
□	UnitName1	=	varchar
□	UnitInd2	=	varchar
□	UnitName2	=	varchar
□	UnitInd3	=	varchar
□	UnitName3	=	varchar
□	Author	=	varchar
□	PhyloSortSequence	=	tinyint
□	Status	=	varchar
□	Source	=	varchar
□	Notes	=	varchar
□	Hybrid	=	varchar
□	SecurityStatus	=	int
□	modifiedUid	=	int
□	modifiedTimeStamp	=	datetime
□	InitialTimeStamp	=	timestamp

Proposed Connector Code:

```
<?php
include_once('C:\xampp\htdocs\Symbiota\config\symbini.php');
include_once('C:\xampp\htdocs\Symbiota\config\dbconnection.php');
$conn = MySQLiConnectionFactory::getCon("write");
// Get all expected values from the passed Array.  Values not found will be set as NULL, which may affect SQL query results.
$SciName = array_key_exists('SciName', $_REQUEST)?$conn->real_escape_string($_REQUEST['SciName']):NULL;


// Modify values to include wildcard for LIKE type comparison(s)
$SciName= '%'.$SciName;

$returnVal = 'false';
$allSuccess = true;
$deletedRows = 0;
$updatedRows = 0;
$conn->autocommit(false);
if(!is_null($SciName))
{
    //==========================================================
    // get values from all necessary tables from the top down
    //==========================================================
    // get TID FROM taxa
    $taxa_TID = Array();
    $stmt = $conn->prepare('SELECT TID FROM taxa WHERE SciName LIKE ? ');

    if (!is_null($stmt) && $stmt != false &&
        $stmt->bind_param("s", $SciName) &&
        $stmt->execute())
    {
        $stmt->bind_result($TID_out);
        while($stmt->fetch())
        {
            $taxa_TID[] = $TID_out;
        }
        $stmt->close();
    }
    else
    {
        $allSuccess = false;
    }

    // get TID, CLID FROM fmchklsttaxalink
    $fmchklsttaxalink_TID_CLID = Array();
    foreach($taxa_TID as $TID)
    {
        $stmt = $conn->prepare('SELECT TID, CLID FROM fmchklsttaxalink WHERE fmchklsttaxalink.TID = ?');
        if (!is_null($stmt) && $stmt != false &&
            $stmt->bind_param("i", $TID) &&
            $stmt->execute())
        {
            $stmt->bind_result($TID_out, $CLID_out);
```

Figure 3.6: Options for configuring a delete connector and the code generated.

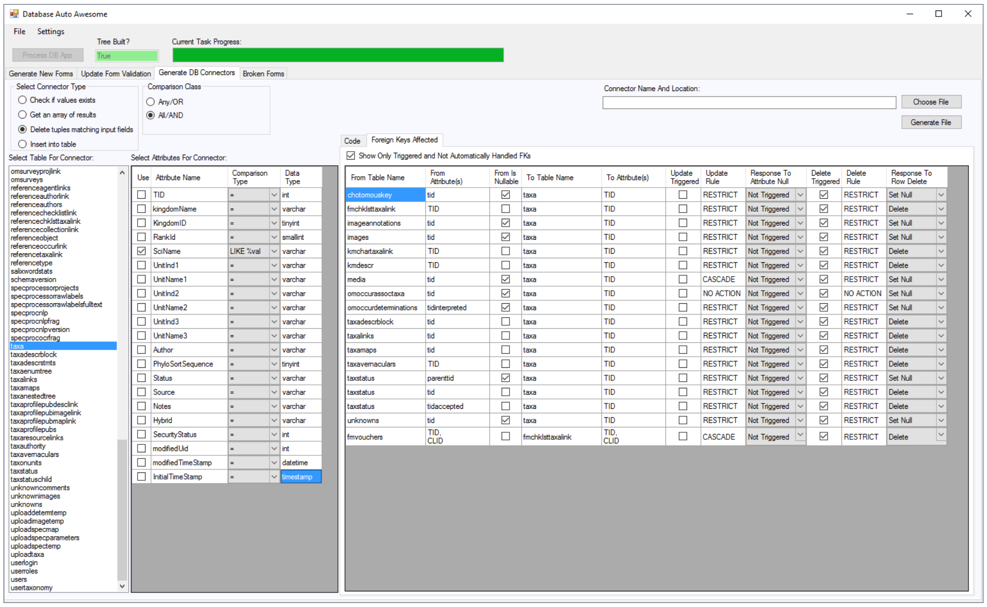
3.5 Generate Form Validation

To have a quality user interface, web forms must check the validity of the data entered by a user into a form. Checking the data is necessary to prevent errors when accessing the database. Checking the data before it reaches the database also allows the form to provide more meaningful feedback to the user about the problems found in their input. If it were to merely return the error given by the database when violating a foreign key constraint, the error would be something like the following.

```
ERROR 1452 (23000): Cannot add or update a child: a foreign key constraint
fails ('namesinc'_'employee', CONSTRAINT 'employee-ibfk_1 'FOREIGN KEY
('Dept_ID') REFERENCES 'DEPARTMENTS' ('DEPT_ID'))
```

As a contrasting example, it could check the constraint as part of the validation and return the following message.

```
Values not found: The provided employee department ID must be a valid
department ID
```



Tree Built? True

Current Task Progress:

Select Connector Type
○ Check if values exists
○ Get an array of results
● Delete tuples matching input fields
○ Insert into table

Comparison Class
○ Any/OR
● All/AND

Connector Name And Location:

Select Table For Connector:

omsurveyprojlink
omsurveys
referenceagentlinks
referenceauthorlink
referenceauthors
referencechecklistlink
referencechklsttaxalink
referencecollectionlink
referenceobject
referenceoccurlink
referencetaxalink
referencetype
salixwordstats
schemaversion
specprocessorprojects
specprocessorrawlabels
specprocessorrawlabelsfulltext
specprocnlp
specprocnlpfrag
specprocnlpversion
specprococrfrag
taxa
taxadescrblock
taxadescrstmts
taxaenumtree
taxalinks
taxamaps
taxanestedtree
taxaprofilepubdesclink
taxaprofilepubimagelink
taxaprofilepubmaplink
taxaprofilepubs
taxaresourcelinks
taxauthority
taxavernaculars
taxonunits
taxstatus
taxstatuschild
unknowncomments
unknownimages
unknowns
uploaddetermtemp
uploadimagetemp
uploadspecmap
uploadspecparameters
uploadspectemp
uploadtaxa
userlogin
userroles
users
usertaxonomy

Select Attributes For Connector:

Use	Attribute Name	Comparison Type	Data Type
□	TID	=	int
□	kingdomName	=	varchar
□	KingdomID	=	tinyint
□	RankId	=	smallint
☑	SciName	LIKE %val	varchar
□	UnitInd1	=	varchar
□	UnitName1	=	varchar
□	UnitInd2	=	varchar
□	UnitName2	=	varchar
□	UnitInd3	=	varchar
□	UnitName3	=	varchar
□	Author	=	varchar
□	PhyloSortSequence	=	tinyint
□	Status	=	varchar
□	Source	=	varchar
□	Notes	=	varchar
□	Hybrid	=	varchar
□	SecurityStatus	=	int
□	modifiedUid	=	int
□	modifiedTimeStamp	=	datetime
□	InitialTimeStamp	=	timestamp

☑ Show Only Triggered and Not Automatically Handled FKs

From Table Name	From Attribute(s)	From Is Nullable	To Table Name	To Attribute(s)	Update Triggered	Update Rule	Response To Attribute Null	Delete Triggered	Delete Rule	Response To Row Delete
chotomouskey	tid	☑	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Set Null
fmchklsttaxalink	TID	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
imageannotations	tid	☑	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Set Null
images	tid	☑	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Set Null
kmchartaxalink	TID	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
kmdescr	TID	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
media	tid	☑	taxa	TID	□	CASCADE	Not Triggered	☑	RESTRICT	Set Null
omoccurassoctaxa	tid	☑	taxa	TID	□	NO ACTION	Not Triggered	☑	NO ACTION	Set Null
omoccurdeterminations	tidinterpreted	☑	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Set Null
taxadescrblock	tid	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
taxalinks	tid	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
taxamaps	tid	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
taxavernaculars	TID	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
taxstatus	parenttid	☑	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Set Null
taxstatus	tid	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
taxstatus	tidaccepted	□	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Delete
unknowns	tid	☑	taxa	TID	□	RESTRICT	Not Triggered	☑	RESTRICT	Set Null
fmvouchers	TID, CLID	□	fmchklsttaxalink	TID, CLID	□	CASCADE	Not Triggered	☑	RESTRICT	Delete

Figure 3.7: Foreign key constraints that must be handled manually by the delete connector.

DBAA provides two ways of adding form validation to a form: as part of the new form generation process, which will be discussed later, and independently to existing forms. During the preprocessing stage, DBAA identifies all forms found in the DBApp. By identifying the code used to submit the form, it also traces the series of function calls made and finds all SQL statements performed as a result of that form being submitted. It then attempts to match the individual form fields to columns in the database by connecting the columns and tables used in the SQL statements to the submitted field ids used to form the SQL statements.

Using the database columns used by the form DBAA can again refer to the information it gathered and identify which form validators are applicable to the form, based on data types, nullability, unique constraints, and foreign key constraints. With this information, it provides a user interface where the user can select an existing form from a list, and be presented with a list of all applicable form validators that can be added to the form. In the interface, DBAA shows the form's existing HTML and JavaScript code. It also adds in the validator code to match what the revised version would look like and highlights the changes in red. That way the user has an immediate preview of all changes that will occur from the addition of the selected validators.

For any of the script validators, DBAA adds an HTML tag to the form where the validator can place an error message if the input fails validation. This gives the validators a way to provide cleaner and more relevant error messages.

There are three main types of input validation that typically occur in a form: HTML restriction attributes, single field on change scripts, and form submission scripts. HTML restriction attributes are attributes included in the input tag of the form that impose a restriction on what can be entered in that field. The primary attribute used is the *maxlength* attribute, which allows you to specify the maximum number of characters that can be entered into the field. HTML 5 added some additional attributes, like *min* and *max*, which allow you to set the minimum and maximum values of a number entered into the field; and *pattern*, which allows you to set a regular expression that the value in the field must match.

Single field on change scripts are validation scripts that are called when the value in a field has been changed. Since a script is being called, the validation can be anything a script can check. In DBAA, this type of script is primarily used as a data type validation script. To perform a data type validation, it evaluates the value the user has entered to see if it meets the definition of the data type for that field from the database schema. For number values, it can check that the numbers are valid integers or non-integers as required by the schema, and that the value falls within the accepted range. For string data types, it can check that the value is no longer than what is allowed by the schema.

Form submission scripts are validation scripts that are executed when a user submits the form, and check one or more field. This validation is the last that happens before the form input values are sent to the back end to be used in database operations. DBAA uses this type of form validation to check three conditions. In all cases, if a failure is detected, an appropriate error message is shown, and the form submission does not proceed.

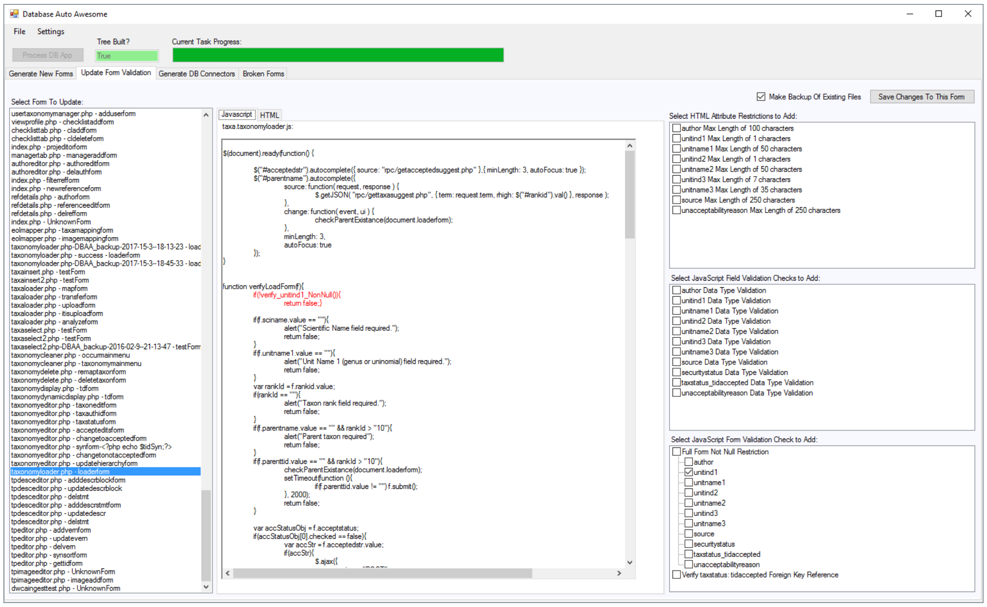
First, DBAA uses the form submission script to check that required fields are not empty. This validation takes the values of all required fields and checks if any of them are null, or empty. All fields marked as required must have a value provided, or the entire check is considered failed.

Second, DBAA uses the form submission script to check that fields that must be unique are actually unique. Using the database schema information, DBAA can identify fields or groups of fields that must have all unique entries within the table. On submitting the form, it takes the values entered in these fields and sends them to a database connector that checks for the existence of those same values in the table. If the connector returns true, it knows that the field values are not unique and cannot be used for that operation, so the check is considered failed.

Third, DBAA uses the form submission script to check foreign key constraints on values that are being inserted into the table. This is similar to the uniqueness check. In this case, it checks for the existence of those values in a table other than the one it is inserting into and if the response is true, it knows that the values being referenced exist and it can properly insert these values into the table. If they are not found, then it knows that it cannot insert the values into the table and the check is considered failed.

Each of the three main types of validators is implemented as a class. Each validator is a subclass to its type class. Each validator class is responsible for providing the code necessary for it to function. Additional validators can be added by creating new subclasses and providing the necessary code generation function and the code for identifying when each check type is applicable. DBAA compiles a list of all applicable validators and when it is time to generate the code, it gets the code from each instance of each class for those validators that were selected by the user. As necessary, such as in the case of the uniqueness or foreign key validators, DBAA also generates the requisite database connector, using the database connector generator.

To update the validation scripts of an existing form, DBAA provides a simple UI as shown in Figure 3.8. On the left side is a list of all existing forms that DBAA found in the DBApp. The user selects the form to enhance from that list. Then in the center, there are tabs to show the HTML code of the form and the JavaScript functions called by the form. The existing code will be shown in black and the pending updates will be shown in red. On



Tree Built? True

Current Task Progress:

Select Form To Update:

usertaxonomymanager.php - adduserform
viewprofile.php - checklistaddform
checklisttab.php - claddform
checklisttab.php - cldeleteform
index.php - projeditorform
managertab.php - manageraddform
authoreditor.php - authoreditform
authoreditor.php - delauthform
index.php - filterrefform
index.php - newreferenceform
refdetails.php - authorform
refdetails.php - referenceeditform
refdetails.php - delrefform
index.php - UnknownForm
eolmapper.php - taxamappingform
eolmapper.php - imagemappingform
taxonomyloader.php-DBAA_backup-2017-15-3--18-13-23 - load
taxonomyloader.php - success - loaderform
taxonomyloader.php-DBAA_backup-2017-15-3--18-45-33 - load
taxainsert.php - testForm
taxainsert2.php - testForm
taxaloader.php - mapform
taxaloader.php - transferform
taxaloader.php - uploadform
taxaloader.php - itisuploadform
taxaloader.php - analyzeform
taxaselect.php - testForm
taxaselect2.php - testForm
taxaselect2.php-DBAA_backup-2016-02-9--21-13-47 - testForm
taxonomycleaner.php - occurmainmenu
taxonomycleaner.php - taxonomymainmenu
taxonomydelete.php - remaptaxonform
taxonomydelete.php - deletetaxonform
taxonomydisplay.php - tdform
taxonomydynamicdisplay.php - tdform
taxonomyeditor.php - taxoneditform
taxonomyeditor.php - taxauthidform
taxonomyeditor.php - taxstatusform
taxonomyeditor.php - accepteditsform
taxonomyeditor.php - changetoacceptedform
taxonomyeditor.php - synform-<?php echo $tidSyn;?>
taxonomyeditor.php - changetonotacceptedform
taxonomyeditor.php - updatehierarchyform
taxonomyloader.php - loaderform
tpdesceditor.php - adddescrblockform
tpdesceditor.php - updatedescrblock
tpdesceditor.php - delstmt
tpdesceditor.php - adddescrstmtform
tpdesceditor.php - updatedescr
tpdesceditor.php - delstmt
tpeditor.php - addvernform
tpeditor.php - updatevern
tpeditor.php - delvern
tpeditor.php - synsortform
tpeditor.php - gettidform
tpimageeditor.php - UnknownForm
tpimageeditor.php - imageaddform
dwcaingesttest.php - UnknownForm

taxa.taxonomyloader.js:

```
$(document).ready(function() {

        $("#acceptedstr").autocomplete({ source: "rpc/getacceptedsuggest.php" },{ minLength: 3, autoFocus: true });
        $("#parentname").autocomplete({
                source: function( request, response ) {
                        $.getJSON( "rpc/gettaxasuggest.php", { term: request.term, rhigh: $("#rankid").val() }, response );
                },
                change: function( event, ui ) {
                        checkParentExistance(document.loaderform);
                },
                minLength: 3,
                autoFocus: true
        });
}


function verifyLoadForm(f){
        if(!verify_unitind1_NonNull()){
                return false;}

        if(f.sciname.value == ""){
                alert("Scientific Name field required.");
                return false;
        }
        if(f.unitname1.value == ""){
                alert("Unit Name 1 (genus or uninomial) field required.");
                return false;
        }
        var rankId = f.rankid.value;
        if(rankId == ""){
                alert("Taxon rank field required.");
                return false;
        }
        if(f.parentname.value == "" && rankId > "10"){
                alert("Parent taxon required");
                return false;
        }
        if(f.parenttid.value == "" && rankId > "10"){
                checkParentExistance(document.loaderform);
                setTimeout(function (){
                        if(f.parenttid.value != "") f.submit();
                }, 2000);
                return false;
        }

        var accStatusObj = f.acceptstatus;
        if(accStatusObj[0].checked == false){
                var accStr = f.acceptedstr.value;
                if(accStr){
                        $.ajax({
```

☑ Make Backup Of Existing Files

Select HTML Attribute Restrictions to Add:

☐ author Max Length of 100 characters
☐ unitind1 Max Length of 1 characters
☐ unitname1 Max Length of 50 characters
☐ unitind2 Max Length of 1 characters
☐ unitname2 Max Length of 50 characters
☐ unitind3 Max Length of 7 characters
☐ unitname3 Max Length of 35 characters
☐ source Max Length of 250 characters
☐ unacceptabilityreason Max Length of 250 characters

Select JavaScript Field Validation Checks to Add:

☐ author Data Type Validation
☐ unitind1 Data Type Validation
☐ unitname1 Data Type Validation
☐ unitind2 Data Type Validation
☐ unitname2 Data Type Validation
☐ unitind3 Data Type Validation
☐ unitname3 Data Type Validation
☐ source Data Type Validation
☐ securitystatus Data Type Validation
☐ taxstatus_tidaccepted Data Type Validation
☐ unacceptabilityreason Data Type Validation

Select JavaScript Form Validation Check to Add:

☐ Full Form Not Null Restriction
  - ☐ author
  - ☑ unitind1
  - ☐ unitname1
  - ☐ unitind2
  - ☐ unitname2
  - ☐ unitind3
  - ☐ unitname3
  - ☐ source
  - ☐ securitystatus
  - ☐ taxstatus_tidaccepted
  - ☐ unacceptabilityreason

☐ Verify taxstatus: tidaccepted Foreign Key Reference

Figure 3.8: Update form validation UI.

the right side there are lists of the validators that can be added to the form. After selecting the form, the user selects which validators to add and clicks the save button. By default, DBAA will save a backup of each file being modified.

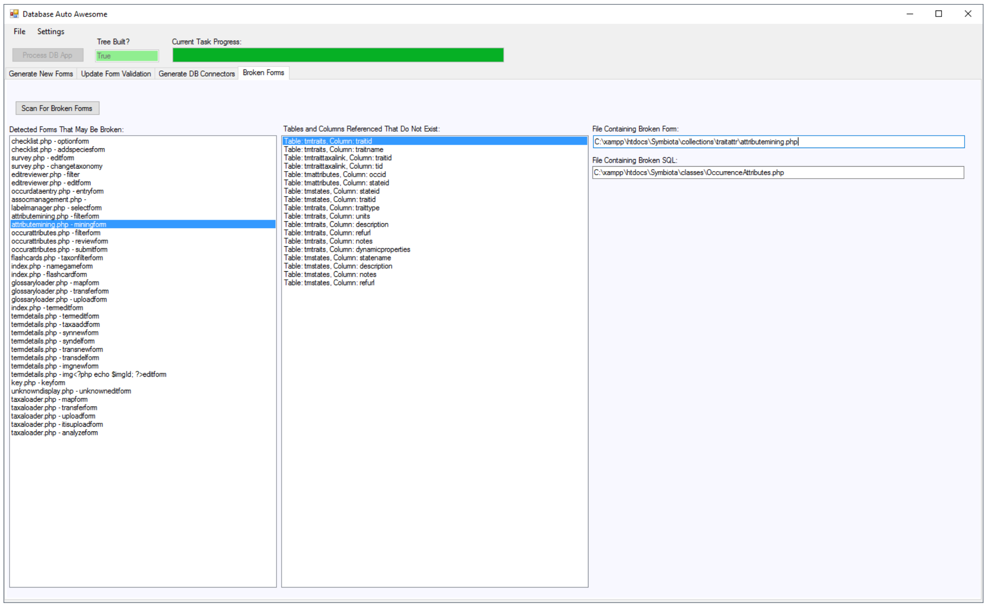
3.6 Identify Forms With Invalid Database Field References

As part of enhancing the set of forms in the DBApp, it is important to identify cases where an existing form is broken so that a new form can be built to replace it. For the purposes of DBAA, the focus is on the information gathered during the preprocessing stage to help identify those forms. In DBAA, a form is considered broken, if, as a result of the form being used anywhere down the line of called code, an SQL column or table is referenced that does not actually exist.

This will not identify forms which are broken due to bugs in the HTML, JavaScript, or PHP, but provides meaningful feedback that is directly related to DBAA's goal of enhancing a set of forms. To identify which forms are broken, it looks at all PHP functions that get called as a result of a given form being submitted. Due to the normal flow control, these functions are not guaranteed to be called in practice, since the run time data affects which functions are called. So, this is just the collection of all functions that could possibly be called, given the correct conditions. If any of the functions found in that collection contain SQL statements that make references to tables or columns that do not exist in the database schema, the form resulting in the function call is marked as being potentially broken.

DBAA provides a list of forms that it has identified as potentially broken, as shown in Figure 3.9. To use this tool, the user clicks the button to scan for broken forms. The left column populates with forms that may be broken. When a form is selected, the center column populates with a list of table and column pairs that are referenced, but don't exist in the database schema. Selecting one of these shows the location of the broken form file and the PHP file that contains the erroneous SQL statement.

In the case of Symbiota, the SQL statements are often built by concatenating together a large number of string literals and variables. In some cases, even the tables being accessed by the query are just set at run time from a variable. Different portions of SQL are



Detected Forms That May Be Broken:

```
checklist.php - optionform
checklist.php - addspeciesform
survey.php - editform
survey.php - changetaxonomy
editreviewer.php - filter
editreviewer.php - editform
occurdataentry.php - entryform
assocmanagement.php -
labelmanager.php - selectform
attributemining.php - filterform
attributemining.php - miningform
occurattributes.php - filterform
occurattributes.php - reviewform
occurattributes.php - submitform
flashcards.php - taxonfilterform
index.php - namegameform
index.php - flashcardform
glossaryloader.php - mapform
glossaryloader.php - transferform
glossaryloader.php - uploadform
index.php - termeditform
termdetails.php - termeditform
termdetails.php - taxaaddform
termdetails.php - synnewform
termdetails.php - syndelform
termdetails.php - transnewform
termdetails.php - transdelform
termdetails.php - imgnewform
termdetails.php - img<?php echo $imgId; ?>editform
key.php - keyform
unknowndisplay.php - unknowneditform
taxaloader.php - mapform
taxaloader.php - transferform
taxaloader.php - uploadform
taxaloader.php - itisuploadform
taxaloader.php - analyzeform
```

Tables and Columns Referenced That Do Not Exist:

```
Table: tmtraits, Column: traitid
Table: tmtraits, Column: traitname
Table: tmtraittaxalink, Column: traitid
Table: tmtraittaxalink, Column: tid
Table: tmattributes, Column: occid
Table: tmattributes, Column: stateid
Table: tmstates, Column: stateid
Table: tmstates, Column: traitid
Table: tmtraits, Column: traittype
Table: tmtraits, Column: units
Table: tmtraits, Column: description
Table: tmtraits, Column: refurl
Table: tmtraits, Column: notes
Table: tmtraits, Column: dynamicproperties
Table: tmstates, Column: statename
Table: tmstates, Column: description
Table: tmstates, Column: notes
Table: tmstates, Column: refurl
```

File Containing Broken Form: C:\xampp\htdocs\Symbiota\collections\traitattr\attributemining.php

File Containing Broken SQL: C:\xampp\htdocs\Symbiota\classes\OccurrenceAttributes.php

Figure 3.9: Potentially broken forms shown in DBAA UI.

often concatenated depending on runtime values of variables. Some portions of the SQL are fetched from functions other than the one building the SQL statement. All of this variability in how a statement is built at run time makes it very difficult to ensure that all invalid table and column references are properly identified. In its current state, DBAA does produce some false positives and likely does not identify all potentially broken forms.

The benefit of identifying forms that may be broken, is that the administrator can then use the DBAA form generator to create a new form to replace the functionality the original was intended to provide. Directly regenerating the form would provide the easiest solution, from a user's point of view, but there are significant barriers to doing so.

The largest barrier stems from the form fields pointing to non-existent database columns. Without any information about what data that field was intended to interact with, it is impossible to know if that data still exists in the database in any form, or how to proceed to rectify the problem. The tool cannot know if the correct solution is to merely remove those fields, to extend the database schema to include the missing columns, or to point the form fields at an alternative set of columns. Automatically taking any action could result in erroneous behavior of the DBApp. So, it is left to the administrators to build new working forms, within the confines of the data that is known to exist.

3.7 Generate New Forms

Creating new forms is a core activity when building, updating, or expanding the functionality of a DBApp. When working on an existing DBApp, it will already contain forms and those forms may be similar in function to what the user wants to create. It will also have a relational database with which the forms interact. These existing portions of the application provide a solid foundation to expand on by generating new forms.

An administrator may find that they need to create new forms to suit the needs of their organization. The ability to use existing information to generate new forms is based on bringing together the parts of the DBAA tool that have been built so far. Generating forms uses the gathered schema and form information, the database connector generator, the form validator generator, and adds in the form generation and connects everything together.

A complete form consists of several parts, as shown in Figure 3.10: an HTML input form, JavaScript form validation and form processing scripts, and the back end PHP code to execute the operation on the database and provide the desired results.

The information the DBAA tool has gathered is everything that is required, from a technical standpoint, to generate the forms. All that remains are the user's specifications for what the form should be.

In the DBAA implementation, the form generation proceeds through several steps, walking the user through each portion.

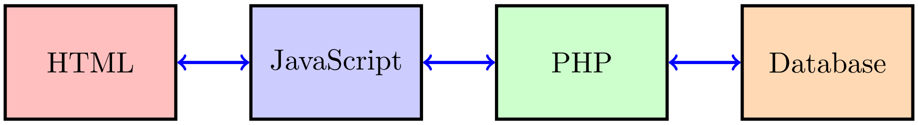


Figure 3.10: Sections of a DBAA generated form.

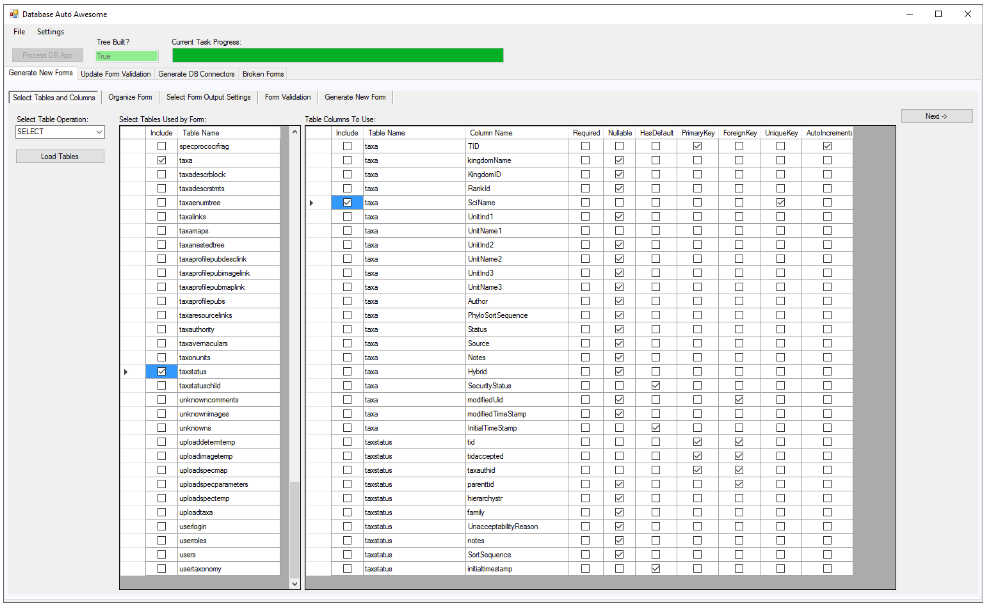
3.7.1 Select form type and target data

The first step is to define what type of interaction the user intends to have with the database, and to specify which data they intend to interact with. In DBAA, the options for interaction type are defined as insert, select, and delete. These three interactions cover the majority of interactions found in Symbiota and establish a large base set of functionalities that can be added or enhanced.

Once the interaction type is specified, the user is provided with a list of the database tables, as shown in Figure 3.11. Selecting a table populates a list of database columns that exist in that table. Due to the nature of such operations, insert and delete operations work on a single table, while select can operate on multiple.

DBAA displays each column with several characteristics of the column stated next to it: whether the field is nullable, whether it auto-increments on insert, whether it is part of a unique key, etc. This allows the user to be immediately informed about the selections they are making. This aids in ensuring all relevant columns are selected, without having to manually refer to the schema to check column properties.

The list of columns allows the user to select which columns they would like the form to



Tree Built? True

Select Table Operation: SELECT

Select Tables Used by Form:

Include	Table Name
□	specprococrfrag
☑	taxa
□	taxadescrblock
□	taxadescrstmts
□	taxaenumtree
□	taxalinks
□	taxamaps
□	taxanestedtree
□	taxaprofilepubdesclink
□	taxaprofilepubimagelink
□	taxaprofilepubmaplink
□	taxaprofilepubs
□	taxaresourcelinks
□	taxauthority
□	taxavernaculars
□	taxonunits
☑	taxstatus
□	taxstatuschild
□	unknowncomments
□	unknownimages
□	unknowns
□	uploaddetermtemp
□	uploadimagetemp
□	uploadspecmap
□	uploadspecparameters
□	uploadspectemp
□	uploadtaxa
□	userlogin
□	userroles
□	users
□	usertaxonomy

Table Columns To Use:

Include	Table Name	Column Name	Required	Nullable	HasDefault	PrimaryKey	ForeignKey	UniqueKey	AutoIncrements
□	taxa	TID	□	□	□	☑	□	□	☑
□	taxa	kingdomName	□	☑	□	□	□	□	□
□	taxa	KingdomID	□	☑	□	□	□	□	□
□	taxa	RankId	□	☑	□	□	□	□	□
☑	taxa	SciName	□	□	□	□	□	☑	□
□	taxa	UnitInd1	□	☑	□	□	□	□	□
□	taxa	UnitName1	□	□	□	□	□	□	□
□	taxa	UnitInd2	□	☑	□	□	□	□	□
□	taxa	UnitName2	□	☑	□	□	□	□	□
□	taxa	UnitInd3	□	☑	□	□	□	□	□
□	taxa	UnitName3	□	☑	□	□	□	□	□
□	taxa	Author	□	☑	□	□	□	□	□
□	taxa	PhyloSortSequence	□	☑	□	□	□	□	□
□	taxa	Status	□	☑	□	□	□	□	□
□	taxa	Source	□	☑	□	□	□	□	□
□	taxa	Notes	□	☑	□	□	□	□	□
□	taxa	Hybrid	□	☑	□	□	□	□	□
□	taxa	SecurityStatus	□	□	☑	□	□	□	□
□	taxa	modifiedUid	□	☑	□	□	☑	□	□
□	taxa	modifiedTimeStamp	□	☑	□	□	□	□	□
□	taxa	InitialTimeStamp	□	□	☑	□	□	□	□
□	taxstatus	tid	□	□	□	☑	☑	□	□
□	taxstatus	tidaccepted	□	□	□	☑	☑	□	□
□	taxstatus	taxauthid	□	□	□	☑	☑	□	□
□	taxstatus	parenttid	□	☑	□	□	☑	□	□
□	taxstatus	hierarchystr	□	☑	□	□	□	□	□
□	taxstatus	family	□	☑	□	□	□	□	□
□	taxstatus	UnacceptabilityReason	□	☑	□	□	□	□	□
□	taxstatus	notes	□	☑	□	□	□	□	□
□	taxstatus	SortSequence	□	☑	□	□	□	□	□
□	taxstatus	initialtimestamp	□	□	☑	□	□	□	□

Figure 3.11: Form type, table, and column selection.

interact with directly. In the case of an insert form, this is the list of columns the form will gather values for. In the cases of select and delete forms, these columns are the columns used as comparisons with existing data to find matching tuples in the data.

For the select and delete types of form, there is also an option to perform an *all* or *any* type comparison. In an all comparison, if all provided values match a given tuple it is included as part of the target data. This is specified in the SQL statement by using *AND* operators between each comparison. In an any comparison, if any provided value matches that column in the tuple, it is included as part of the target data. This is specified in the SQL statement by using *OR* operators between each comparison.

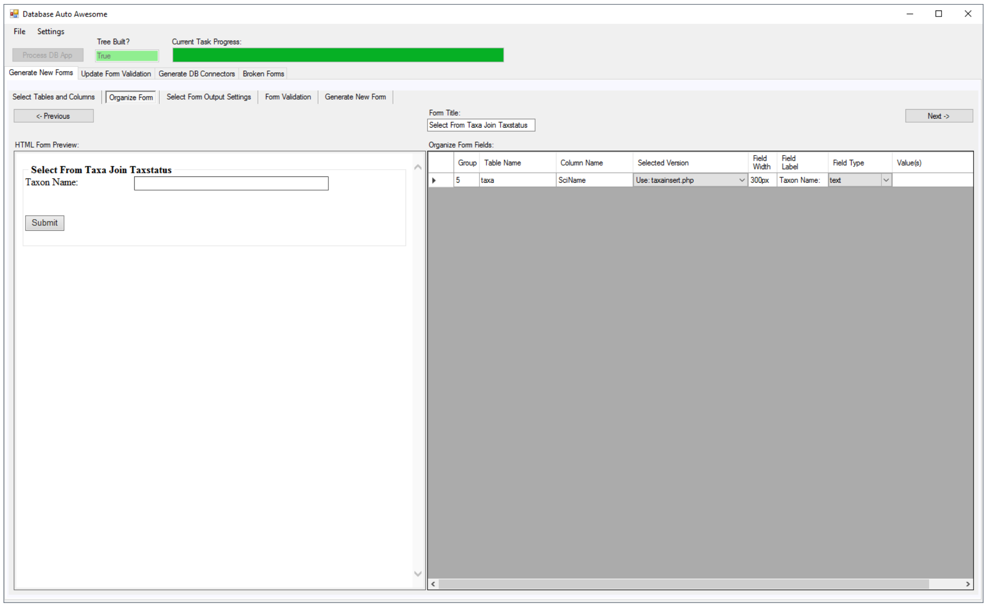
To further aid the user, when building an insert form, values must be provided for all columns that are non-nullable, and do not have a default value designated. So, any column in the selected table that falls into that criteria is marked as required and must be included in the form. This is to prevent the building of forms that will never work.

3.7.2 Form layout

From there, the user is provided with a visual mockup of what the form will look like. This is simply done by generating the HTML required for the form, and displaying it in an embedded web browser panel. The generated version updates immediately as the user makes any changes to the layout configuration. An example of this preview is shown in the left side of Figure 3.12. On the right, there are options to configure the form.

The user can customize the order, look, and type of fields. They can provide the visible labels for the fields, as well as a title for the form. For each field, a selection is offered, to create a new field from scratch, or to copy one from another table that interacts with the same DB field.

This information is available from the initially gathered information, and allows users to use a given existing form as a starting point for building their new form. In cases where an administrator wants to provide a custom version of an existing form in order to simplify it or expand its functionality, or to build a version of a broken form using database fields that



Tree Built? True

Form Title: Select From Taxa Join Taxstatus

Select From Taxa Join Taxstatus

Taxon Name:

	Group	Table Name	Column Name	Selected Version	Field Width	Field Label	Field Type	Value(s)
▸	5	taxa	SciName	Use: taxainsert.php	300px	Taxon Name:	text	

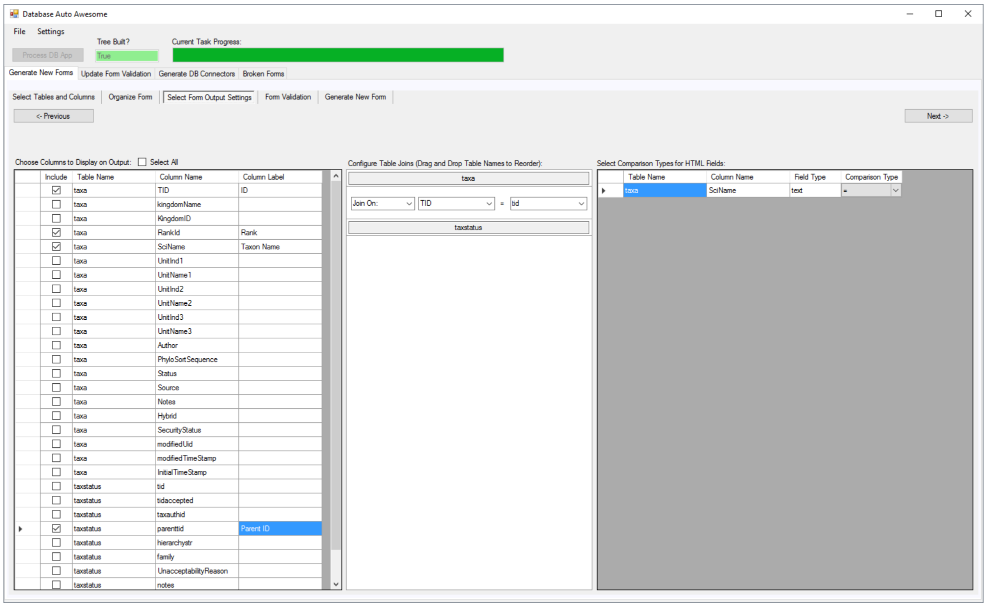
Figure 3.12: Example form preview during form layout step.

actually exist, they can pull the existing field formatting and settings in, without having to configure it themselves.

3.7.3 Configuration

The next step varies based on the type of form being built. For an insert form, this step is skipped all together. For a delete form, the user is presented with the list of comparisons that will be made and the comparator to be used with each field. The user can select from any valid SQL comparator, including LIKE, and set it individually for each field. When generating the SQL statement, the chosen comparator will be inserted into that comparison and function as expected. All the comparators default to the equality comparator. If this is correct for the intended purpose of the form, no changes need to be made.

For a select form, the same comparator selector is shown, but there are two additional sections, like those in Figure 3.13. First, the user can select which columns from the tables selected in the first step should be used as the output columns of the select statement. This allows the user to determine exactly which results they want to view from the table. They can also set the visible name to use for the column when displaying the results.



Tree Built? True

Current Task Progress:

Choose Columns to Display on Output: □ Select All

	Include	Table Name	Column Name	Column Label
	☑	taxa	TID	ID
	□	taxa	kingdomName	
	□	taxa	KingdomID	
	☑	taxa	RankId	Rank
	☑	taxa	SciName	Taxon Name
	□	taxa	UnitInd1	
	□	taxa	UnitName1	
	□	taxa	UnitInd2	
	□	taxa	UnitName2	
	□	taxa	UnitInd3	
	□	taxa	UnitName3	
	□	taxa	Author	
	□	taxa	PhyloSortSequence	
	□	taxa	Status	
	□	taxa	Source	
	□	taxa	Notes	
	□	taxa	Hybrid	
	□	taxa	SecurityStatus	
	□	taxa	modifiedUid	
	□	taxa	modifiedTimeStamp	
	□	taxa	InitialTimeStamp	
	□	taxstatus	tid	
	□	taxstatus	tidaccepted	
	□	taxstatus	taxauthid	
▶	☑	taxstatus	parenttid	Parent ID
	□	taxstatus	hierarchystr	
	□	taxstatus	family	
	□	taxstatus	UnacceptabilityReason	
	□	taxstatus	notes	

Configure Table Joins (Drag and Drop Table Names to Reorder):

taxa

Join On: TID = tid

taxstatus

Select Comparison Types for HTML Fields:

	Table Name	Column Name	Field Type	Comparison Type
▶	taxa	SciName	text	=

Figure 3.13: Options for configuring a select type form.

Second, if more than one table is being used, the user can customize the table joins to produce their desired output in the middle section of the DBAA UI. The user can order the tables and then specify which columns to join on, to form a chain of joins for all included tables. This allows DBAA to know how to generate the database connector used to fetch the results.

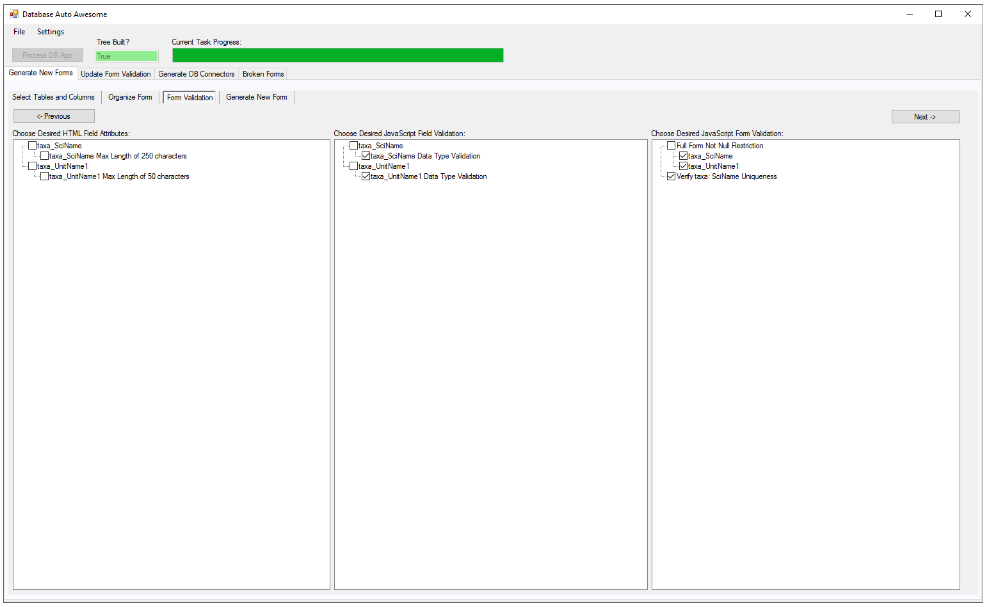
3.7.4 Validation

At this step, DBAA pulls in all the same data used in the standalone validation generator. It generates a list of the applicable form validation types for this form, as shown in Figure 3.14. The user can select all, none, or some of the checks, as desired.

The list of selected form validators will be fed to the form generation tool and the code for the validators will be generated right in line with the rest of the code. This allows DBAA to seamlessly add or remove the validators as the user chooses.

3.7.5 Generation

Upon reaching the last step, the user is presented with the generated HTML, JavaScript,



Tree Built? True

Current Task Progress:

Choose Desired HTML Field Attributes:

- □ taxa_SciName
  - □ taxa_SciName Max Length of 250 characters
- □ taxa_UnitName1
  - □ taxa_UnitName1 Max Length of 50 characters

Choose Desired JavaScript Field Validation:

- □ taxa_SciName
  - ☑ taxa_SciName Data Type Validation
- □ taxa_UnitName1
  - ☑ taxa_UnitName1 Data Type Validation

Choose Desired JavaScript Form Validation:

- □ Full Form Not Null Restriction
  - ☑ taxa_SciName
  - ☑ taxa_UnitName1
- ☑ Verify taxa: SciName Uniqueness

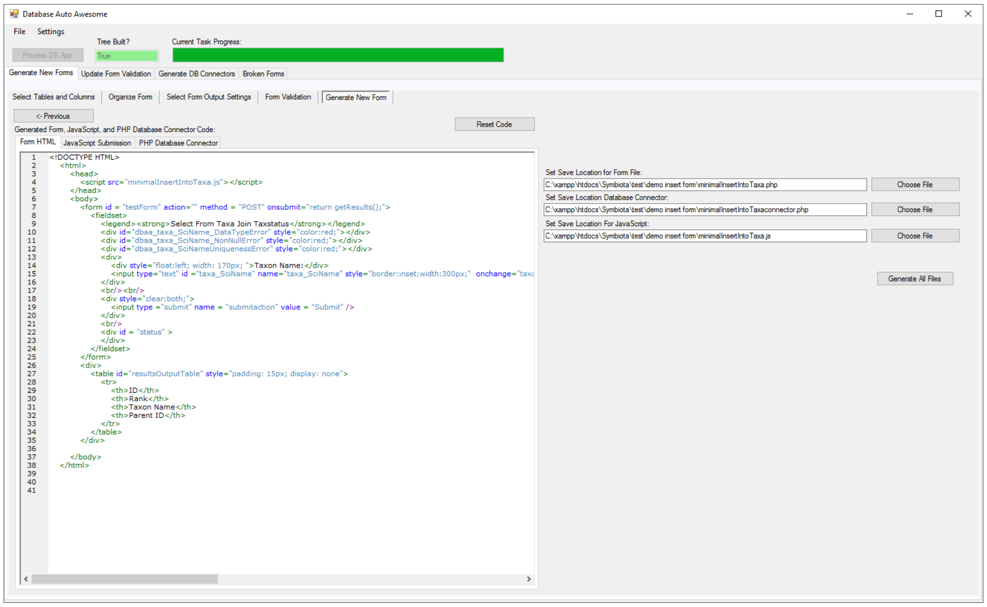
Figure 3.14: New form validation options.

and PHP code needed for this form, as shown in the left side of Figure 3.15. It is shown here primarily to demonstrate that it has completed the task of code generation. For a more advanced user, this may be beneficial for them to be able to verify the code is what they desired, before finalizing the code generation.

Here, the user chooses where they want the code to be generated and clicks the generate button. All necessary files will be generated and saved, ready for use or further integration into the system. There will be a minimum of three files generated: the HTML form file, the JavaScript form processing and validation file, and the PHP file for performing the desired database interaction. Additional PHP files will be generated for each form validator that requires one.

The code generation proceeds one file at a time. For each file, it calls on the appropriate generation tool, and passes to it the relevant existing system information and the parameters chosen by the user.

Every portion of this process is supported and informed by the data that has been gathered from the existing project. Without it, the tool would not be able to generate a complete working form. The amount of information that is required from the user is not



Tree Built?: True

Generated Form, JavaScript, and PHP Database Connector Code:

```
<!DOCTYPE HTML>
   <html>
      <head>
         <script src="minimalInsertIntoTaxa.js"></script>
      </head>
      <body>
         <form id = "testForm" action="" method = "POST" onsubmit="return getResults();">
            <fieldset>
               <legend><strong>Select From Taxa Join Taxstatus</strong></legend>
               <div id="dbaa_taxa_SciName_DataTypeError" style="color:red;"></div>
               <div id="dbaa_taxa_SciName_NonNullError" style="color:red;"></div>
               <div id="dbaa_taxa_SciNameUniquenessError" style="color:red;"></div>
               <div>
                  <div style="float:left; width: 170px; ">Taxon Name:</div>
                  <input type="text" id ="taxa_SciName" name="taxa_SciName" style="border:inset;width:300px;"  onchange="taxa
               </div>
               <br/><br/>
               <div style="clear:both;">
                  <input type ="submit" name = "submitaction" value = "Submit" />
               </div>
               <br/>
               <div id = "status" >
               </div>
            </fieldset>
         </form>
         <div>
            <table id="resultsOutputTable" style="padding: 15px; display: none">
               <tr>
                  <th>ID</th>
                  <th>Rank</th>
                  <th>Taxon Name</th>
                  <th>Parent ID</th>
               </tr>
            </table>
         </div>

      </body>
   </html>
```

Set Save Location for Form File: C:\xampp\htdocs\Symbiota\test\demo insert form\minimalInsertIntoTaxa.php

Set Save Location Database Connector: C:\xampp\htdocs\Symbiota\test\demo insert form\minimalInsertIntoTaxaconnector.php

Set Save Location For JavaScript: C:\xampp\htdocs\Symbiota\test\demo insert form\minimalInsertIntoTaxa.js

Figure 3.15: Generated form code and file save locations.

zero, but it takes very little knowledge and work to generate complete forms compared to doing it manually.

Various types of information are fed into each portion of the code generation process, as shown in Figure 3.16. By bringing this information into the process, DBAA can generate completely new or enhanced ways of interacting with the DBApp with minimal guidance from the administrator.

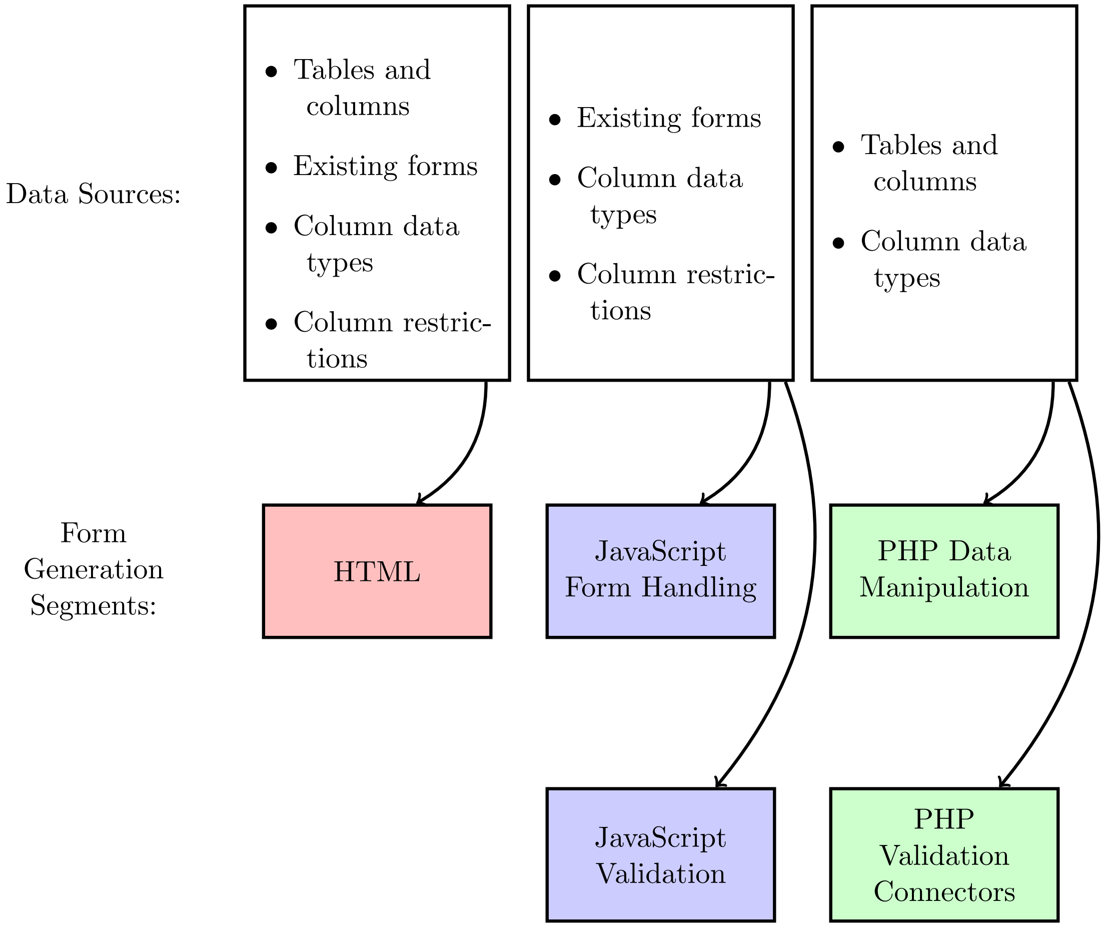


Figure 3.16: Data sources that inform each portion of code generation.

CHAPTER 4
RESULTS

Informed code generation improves the maintainability and extensibility of an application by reducing the knowledge and time required to make modifications to the application. DBAA demonstrates this improvement by providing the tools required by a user to make changes with little effort or knowledge.

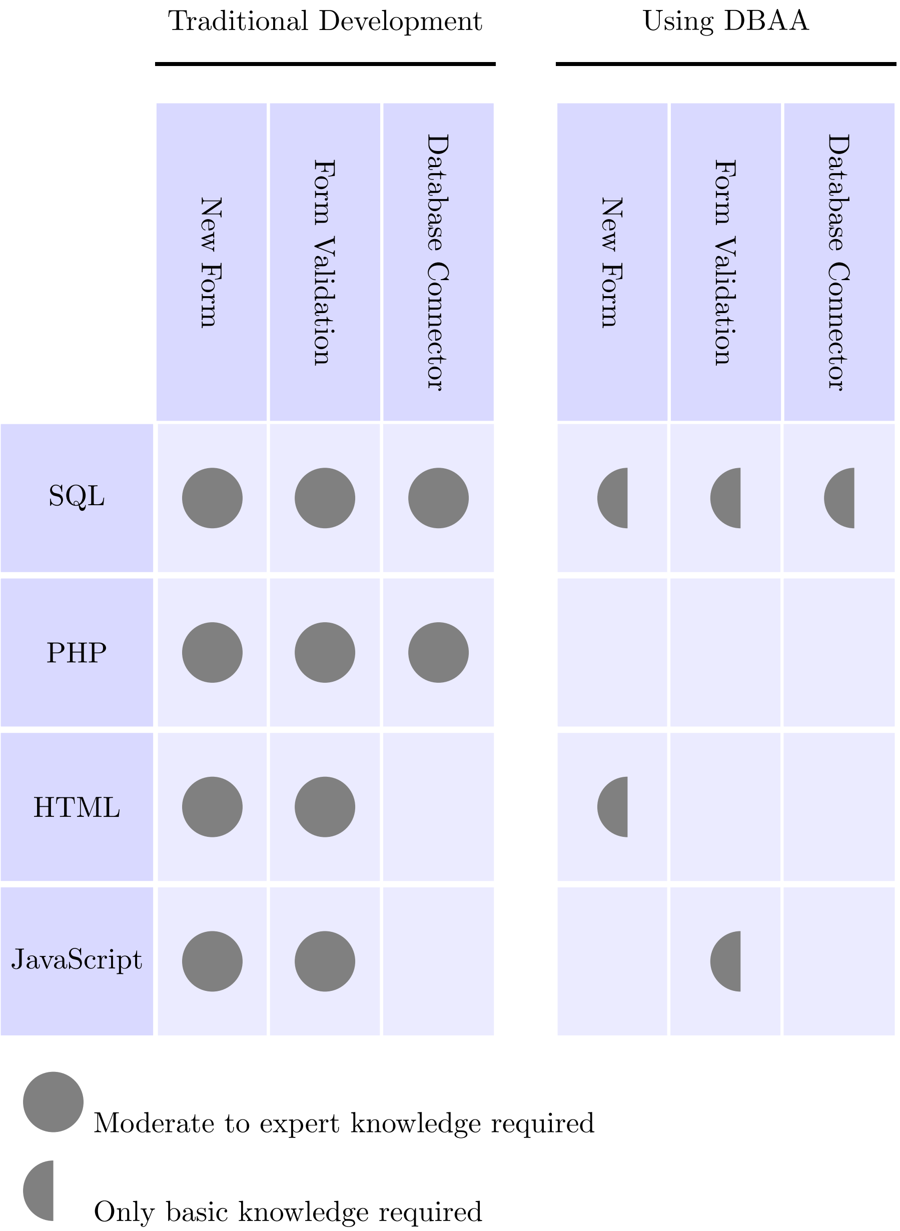
4.1 Reduced Knowledge Required

Symbiota is a complex application and one that is hard for new developers to modify. It uses many different languages and technologies. Because DBAA generates all the code needed to perform modifications to the application, the user does not need to have a working knowledge of these technologies. This stands in contrast to traditional development where the person making changes needs to be capable of writing and interpreting code in all technologies used by the application.

In Figure 4.1 the knowledge required in traditional development and the knowledge required to use DBAA are compared. DBAA removes the requirement to know PHP, the most used technology in Symbiota. It also reduces the SQL knowledge required to a basic understanding of how data is stored in tables and columns and of how comparators are used in SQL statements. This allows the user to target the data they would like to interact with. The user does need to understand how to build SQL statements.

A basic understanding of HTML will aid in formatting and organizing the forms being generated. Some understanding of JavaScript will aid in identifying desired form validation scripts when updating an existing form's validation. No other knowledge is required to operate DBAA.

DBAA does not completely remove the need for an understanding of software development technologies, but it significantly reduces that barrier when compared to traditional development.



	Traditional Development			Using DBAA		
	New Form	Form Validation	Database Connector	New Form	Form Validation	Database Connector
SQL	●	●	●	◖	◖	◖
PHP	●	●	●			
HTML	●	●		◖		
JavaScript	●	●			◖	

● Moderate to expert knowledge required

◖ Only basic knowledge required

Figure 4.1: Knowledge required to perform DBApp modifications in various areas with or without DBAA.

4.2 Reduced Development Time

The tasks performed by DBAA reduce the development time by generating code that would otherwise be written manually. In the example of generating an entirely new form, a user spends a small amount of time filling out a form of options and then DBAA will immediately generate hundreds of lines of code covering PHP, JavaScript, HTML, and SQL. Whether the user is a novice or an experienced developer, this results in a significant reduction in development time compared to writing those hundreds of lines of code manually.

4.3 Improved Accessibility

By reducing the required knowledge and development time to modify a DBApp, DBAA makes it easier for any potential DBApp administrator and their team to modify the application. By lowering these barriers, a greater number of organizations will have sufficient resources to deploy and maintain applications like Symbiota and benefit from the work done to create them.

4.4 Challenges

The initial development effort of a system for generating and updating forms will likely outweigh what would be required for making an administrator's desired changes to the application. For DBAA to be truly beneficial, it must be developed and deployed in a way that will allow it to benefit many different administrators. In the case of a DBApp like Symbiota, such a tool could be built specifically for that application and distributed alongside the application itself. This would spread the development effort across all installed instances of the DBApp, allowing all administrators to benefit.

Generating code requires targeting a specific set of languages, frameworks, and technologies when building the tools. This is another reason why the best approach for creating such a tool may be to build it upfront and release it alongside an open source project. Alternatively, if it were to be built with true modularity in mind and set up as an open source project, a tool like this could be grown to include many combinations of languages, frameworks, and technologies.

The quality of generated code is tied to the person or persons who built the tool. Poor design of the tools will result in inferior quality code being generated by each person using the tool. On the other hand, a well-built tool means that high quality code will be generated for each person using the tool. The key to providing real benefit is to have the code output designed by developers who are experts in the types of code being generated. This allows all administrators to generate code that matches the quality of an expert developer, even if they do it themselves or can only afford to hire a novice developer. This is similar to the findings by Vial [10] in their work to automate schema refactoring. Using their tools, a novice developer could make changes that fit the standards set by their more experienced developers.

In all aspects of building an automated system, like DBAA, there is an act of balancing automation and flexibility. The greater degree of automation, the less flexibility there is for the user to get what they want out of the tool. It becomes necessary to search for a balance that does not require more out of the users than they are capable of, while also not restricting the functionality of the tool to the point where it is no longer substantially useful.

For example, in DBAA, the user can specify which comparators to use for each field of the form. This may be outside the scope of a user's knowledge before using the tool, but removing the option would severely limit the variety of functionality one can get out of the generated forms.

4.5 Future Work

As mentioned in the introduction, there are other areas wherein a DBApp could be automatically or semi-automatically enhanced: creating interfaces to other tools, content management system integration, and database mediation and migration scripts. Work in these areas could yield additional enhancements for reducing the skills required to build unique applications from an existing application.

Taking the informed code generation technique and applying it to companies that build large scale, or a large quantity of, web applications could find benefits in providing tools for developers to automate tasks like form generation. If an organization uses the same set of frameworks and tools for building many forms for a single project, or many similar projects, it could allow the organization to build custom sets of web forms for their customers in a fraction of the time as traditional manual development.

Research into building a more generalized framework that could allow for the interoperation of many different languages and frameworks should be explored. Developing a general-purpose form building tool that could work with any project's framework and back end would save a very large amount of collective development time by reducing how many times developers build nearly identical pieces of code.

CHAPTER 5
CONCLUSION

The skill required to modify or extend a given web application is directly related to how maintainable that application is. Whatever tools reduce the skill required will also improve the maintainability and extensibility of that web application.

In this test implementation, DBAA, we have put into practice informed code generation which allows us to directly extend and modify parts of a web application. It is not as fully automated as Google's Auto Awesome tool is for photos, but it lowers the level of knowledge and skill required to extend or modify the Symbiota application. It enables users without complete knowledge of web application development and the tools used in Symbiota to create new forms and improve validation on existing forms as they see fit. There is immense potential for informed code generation tools to improve the customizability and extensibility of a compatible web application and warrants continued research into these techniques as applied to enhancing existing web applications.

REFERENCES

[1] C. Gries, E. Gilbert, and N. Franz, “Symbiota A virtual platform for creating voucher-based biodiversity information communities,” *Biodiversity Data Journal*, vol. 2, pp. e1114+, Jun. 2014. [Online]. Available: http://dx.doi.org/10.3897/bdj.2.e1114.

[2] W. Frakes and C. Terry, “Software Reuse: Metrics and Models,” *ACM Comput. Surv.*, vol. 28, no. 2, pp. 415–435, Jun. 1996. [Online]. Available: http://dx.doi.org/10.1145/234528.234531.

[3] J.-F. Lépine. PhpMetrics: a static analysis tool for PHP. [computer program]. [Online]. Available: http://www.phpmetrics.org/about.html.

[4] A. Nandi and H. V. Jagadish, “Assisted Querying Using Instant-response Interfaces,” in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’07. New York, NY, USA: ACM, 2007, pp. 1156–1158. [Online]. Available: http://dx.doi.org/10.1145/1247480.1247640.

[5] G. Wolf, H. Khatri, B. Chokshi, J. Fan, Y. Chen, and S. Kambhampati, “Query Processing over Incomplete Autonomous Databases,” in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB ’07. VLDB Endowment, 2007, pp. 651–662. [Online]. Available: http://portal.acm.org/citation.cfm?id=1325926.

[6] I. Ilyas, V. Markl, P. J. Haas, P. G. Brown, and A. Aboulnaga, “Automatic relationship discovery in self-managing database systems,” in *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 2004, pp. 340–341. [Online]. Available: http://dx.doi.org/10.1109/icac.2004.1301405.

[7] M. Jayapandian and H. V. Jagadish, “Automated Creation of a Forms-based Database Query Interface,” *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 695–709, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453932.

[8] SQL Server Migration Assistant. [Online]. Available: https://msdn.microsoft.com/en-us/library/mt613434.aspx.

[9] Y. An, R. Khare, I.-Y. Song, and X. Hu, “Automatically Mapping and Integrating Multiple Data Entry Forms into a Database,” in *Conceptual Modeling ER 2011*, ser. Lecture Notes in Computer Science, M. Jeusfeld, L. Delcambre, and T.-W. Ling, Eds. Springer Berlin Heidelberg, 2011, vol. 6998, pp. 261–274. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24606-7_20.

[10] G. Vial, “Database Refactoring: Lessons from the Trenches,” *IEEE Software*, vol. 32, no. 6, pp. 71–79, Nov. 2015. [Online]. Available: http://dx.doi.org/10.1109/ms.2015.131.

[11] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu, “Making database systems usable,” in *SIGMOD ’07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data.* New York, NY, USA: ACM, 2007, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1145/1247480.1247483.

[12] D. Cedrim, “Context-sensitive Identification of Refactoring Opportunities,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, pp. 827–830. [Online]. Available: http://dx.doi.org/10.1145/2889160.2889266.

[13] T. Sharma, “Identifying Extract-method Refactoring Candidates Automatically,” in *Proceedings of the Fifth Workshop on Refactoring Tools*, ser. WRT ’12. New York, NY, USA: ACM, 2012, pp. 50–53. [Online]. Available: http://dx.doi.org/10.1145/2328876.2328883.

[14] R. S. Xin, W. McLaren, P. Dantressangle, S. Schormann, S. Lightstone, and M. Schwenger, “MEET DB2: Automated Database Migration Evaluation,” *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1426–1434, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1921016.

[15] A. Zisman and J. Kramer, “Information Discovery for Autonomous Database Systems.” [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.3014.