SPECIFICATION AND AUTOMATIC CODE GENERATION OF THE DATA LAYER FOR DATA-INTENSIVE WEB-BASED APPLICATIONS

An Abstract of a Thesis

Submitted

in Partial Fulfillment

of the Requirements for the Degree

Master of Science

Sergei Golitsinski

University of Northern Iowa

May 2008

ABSTRACT

In this thesis I propose a new code generation approach to developing data-intensive web-based applications. I hypothesize that it is possible to build a code generator which will significantly improve development of data-intensive web-based applications by generating at least 50% of the data access code based on a specification of the application's data model. My main argument is that the application's data model is sufficient for deriving most of the data access functionality.

To test this hypothesis I designed a data definition language, came up with a set of rules for deriving data access operations from the application's data model, implemented a code generator, and tested my approach on three applications. The applications used to test the hypothesis included a web-based tool used in criminology for eyewitness identification, a financial reporting system, and a set of web sites for a student association, including a blog, a career web site, several regular web sites and a content management system.

The results were two-fold. On the one hand, the approach proved to be efficient in the sense that 84% - 99% of the data access code was generated automatically – which supported the study's hypothesis. On the other hand, only 20% - 35% of the generated code was actually used by the application. Therefore, considering the amount of generated code, the approach proved to be inefficient. Nevertheless, I maintain that the suggested approach still has the potential for bringing positive results, which may be discovered through further research.

SPECIFICATION AND AUTOMATIC CODE GENERATION OF THE DATA LAYER FOR DATA-INTENSIVE WEB-BASED APPLICATIONS

A Thesis

Submitted

in Partial Fulfillment

of the Requirements for the Degree

Master of Science

Sergei Golitsinski

University of Northern Iowa

May 2008

This Study by: Sergei Golitsinski

Entitled: Specification and Automatic Code Generation of the Data Layer for Data-Intensive Web-Based Applications

has been approved as meeting the thesis requirement for the

Degree of Master of Science

_____	_____
Date	Dr. V. Eugene Wallingford, Chair, Thesis Committee
_____	_____
Date	Dr. Kevin C. O'Kane, Thesis Committee Member
_____	_____
Date	Dr. Douglas J. Shaw, Thesis Committee Member
_____	_____
Date	Dr. Sue A. Joseph, Interim Dean, Graduate College

I dedicate this thesis to my friends and collegues at the UNI Advancement Technology Team – for their infinite patience with me and my coding experiments.

ACKNOWLEDGMENTS

I thank Dr. Eugene Wallingford, my academic advisor in computer science, who taught me that ceilings are movable and never gave up on me, despite the plenty opportunities I offered. I also thank Dr. Kevin O'Kane and Dr. Doug Shaw, my thesis committee members, who helped me understand many of the concepts, which became the foundation of this research.

TABLE OF CONTENTS

PAGE

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Subject of the Study and Motivation

In this study I propose a new approach to developing data-intensive web-based applications, which is based on automatic code generation. I examine this approach to determine whether it simplifies and improves the development process of these applications.

The motivation for this thesis is based on my personal experience as a web developer. Having implemented numerous data-driven web-based applications, I noticed multiple recurring patterns in their implementation, which demonstrated a significant amount of repetitive work. Examining current research in this area revealed that researchers and developers alike are faced with a similar problem: many scholars agree that “web developers spend a significant amount of time for the construction of typical, standardized software modules” (Milosavljevic, Vidakovic, & Konjovic, 2002, p. 1).

Thus, the primary motivation for this study became the search for a way to simplify the development process of data-intensive web-based applications.

Research Hypothesis

This study examines the concept of specifying and automatically generating the code which is used for accessing the data, located in the system’s data storage component, such as a relational database.

A review of current research and similar code generation systems has shown that most approaches require the developer to specify the data access functionality in detail –

i.e., each operation in particular. The main argument of this thesis is that, contrary to existing research, specifying the data model of the application using a modeling approach similar to the entity-relationship model (Chen, 1976), is enough for automatically generating most of the required data access functionality.

The data access functionality, generated based on the data model alone, certainly, will not be an exhaustive listing of all possible operations on this data; however, in my opinion, it will represent the most commonly used functionality in data-intensive web-based applications.

Therefore, I hypothesize that it is possible to build a code generator which will significantly improve development of data-intensive web-based applications by generating at least 50% of the data access code based on a specification of the application's data model. To test this hypothesis I have built such a code generator and used it in developing the data layer for several "real world" applications.

Significance of the Thesis

This thesis offers the following contributions to current research:

1. A code generator which has been successfully tested and is currently used in web application development to generate data access code for the Microsoft .Net / SQL Server platform. The generator produces SQL code as database-level code and in c# or VB.Net as application-level code.

2. The data definition language used to describe the data model of an application is a simplified and modified version of the entity-relationship model and has been successfully tested with the code generator.

3. Generating data access code without explicit specification is, to my best knowledge, a novel approach in this field.

Structure of the Thesis

In Chapter 2, I will provide an overview of the research problem: I will briefly describe data-intensive web-based applications, as well as some implementation issues, caused by recurring functionality requirements. I will review automatic code generation as a possible solution to these issues, after which I will examine several existing code generators and approaches to modeling the data layer of such applications.

In Chapter 3, I will describe the study's methodology. I will discuss the target application's architecture, after which I will define the scope of the code to be generated through examining the data layer implementation and abstracting common functionality. I will conclude with describing the experiment itself, which includes implementation tasks, such as describing the data model, defining rules for data access methods, and constructing the code generator; and testing tasks, such as applying the system to generating code for real applications and measuring the results of using this approach.

In Chapter 4, I will describe the results of the implementation and testing tasks listed in the previous chapter. I will introduce the simple XML-based data definition language I created to describe the application's data model, after which I will define the common data access methods generated by my system. After providing a brief overview on the code generator's implementation and its process model, I will examine the experimental part, including numeric data on how useful the code generator proved to be in terms of the amount of generated code and how much of that code was actually used

by the application.

Chapter 5 will interpret the results of the experiment. I will discuss the major findings, and whether this code generator improved the development process of data-intensive web-based applications. I will also discuss the lessons I have learned through this experiment, which include a review of limitations of this code generation approach and the numerous tradeoffs I had to make, as well possibilities for future improvements of such systems and research in this area.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, I will provide an overview of the research problem: I will describe data-intensive web-based applications, as well as some implementation issues, caused by recurring functionality requirements in such applications. I will show that automatic code generation is a possible solution to these issues, after which I will explain why it is reasonable to generate only part of the application, and why the data access functionality may be the optimal choice. I will also provide a critical review of several existing code generators and approaches to modeling data access functionality.

Definition of Data-Intensive Web-Based Applications

The subject of this thesis is data-intensive web-based applications. There have been many definitions offered which separate data-intensive applications from other types of web-based applications, or web sites. Fratenali and Paolini (2000) define these applications as characterized by high volumes of data to be published and maintained over time (p. 325). According to Merialdo and Atzeni (2000), data-intensive web sites are large sites based on a back-end database with a fairly complex hypertext structure (p. 50). According to Jacob, Schwarz, Kaiser, and Mitschang (2006a), data-intensive web sites mainly focus on making large amounts of data available on the web (p. 77). Similar to this definition, Ceri, Fratenali, and Matera (2002) suggest that these applications' main purpose is presenting large amount of data to their users (p. 20). Other researches offer similar definitions.

While all of these definitions correctly describe this type of application, using such criteria as “large/high volume amounts of data,” or “complex hypertext structure” may be misleading. For example, a web site, displaying its content on a single web page can be, in fact, data-intensive – provided that this content is stored in and retrieved from a database – although it does not present large amounts of data to the user. On the other hand, a web site containing thousands of static HTML pages, clearly, presents large amounts of data to the user; however, I suppose, it cannot be described as data-intensive, since this data is not retrieved from a database or generated dynamically and, therefore, does not require any data-related operations. The amount of data also cannot be used as a qualifier, for it is hard to define a boundary between intensive and not intensive in terms of bytes, characters, or web pages.

Still, these applications clearly stand out from the rest. They are not composed of pre-coded static HTML pages: their content is served from a data repository – which may or may not be a database – with web pages generated dynamically. This causes a significant part of the application’s code to be designated to support this requirement – which justifies, in my opinion, the usage of the relative amount of data access code as a qualifier for such systems. The three applications used in this study have between 19% and 38% of their code handling data access; although, it has been my experience that data access code can represent between 10% and 80% of the application’s code. Considering these numbers, I conclude that for an application to be considered data-intensive, at least 10% of its code should handle data access.

Therefore, for the purposes of this research, I will define data-intensive web-based applications as systems, which require comprehensive data access functionality for providing web-based "read" or "read/write" access to data stored in a data repository, such as a database, with at least 10% of the application's code handling this functionality.

Automatic Code Generation as a Way to Improve Development

The implementation of data-intensive web-based applications contains numerous recurring functionality requirements, which lead to repetitive coding patterns and, often, duplicate code.

One solution to these issues is to factor out as much common functionality as possible into abstract classes, which can be reused, with concrete classes implementing application-specific details; this approach, potentially, can minimize the amount of repetitive work and duplicate code. However, experience has shown that every new application has its own unique requirements, which leads to updating the abstract classes with new code each time they are reused. As a result, the abstract code, while catering to "the most common denominator," very soon becomes unnecessary complex, and ends up supporting numerous functionality requirements, only a small part of which are used by any single application.

A better approach is to automate development by generating the repetitive part of the code. Although generated code contains numerous repetitive elements, it does not lead to any of the problems caused by code duplication: any edits are made to the specification, whereas the code itself is never manually altered.

Benefits of Automatic Code Generation

Automatic code generation offers numerous benefits. According to Jacob et al. (2006a), “creating the specifications of an application is much less time consuming than creating the equivalent code” (p. 78). Whitehead, Ge, and Pan (2004) noted that code generation reduces monotonous work: programmers only need to build models and leave the tedious coding to the generator; “this enables developers to focus on more important development efforts like the domain logic and user interface design” (p. 205). These development efforts can be considered more important simply because they cannot be automatically designed by a machine. In addition to this, by using automation, developers avoid the error-prone process of manual refactoring. Whitehead et al. observed: “to add new features you simply change the model and regenerate the code; then make minimum modifications to the presentation layer” (p. 205). Cleaveland (n.d.) offered a detailed overview of the benefits of using a code generator:

1. Specification Level versus Code Level … Specifications are much easier to read, write, edit, debug, and understand than the code that implements the specification…
2. Separation of Concerns: all too often software is constructed with different concerns all jumbled together… Generators provide a way for separating [these] concerns.
3. Multiple Products: In … typical situations, many other files [in addition to the code itself] may also be generated, including [documentation, test scripts, diagrams, and simulation tools.]
4. Consistency of Information… All too often, fixing a bug or updating software introduces other errors, because a “piece” of information was not updated consistently across the whole product. In a software-generation approach, one simply updates the specification and regenerates the software.
5. Correctness of Generated Products: Many program generators create thousands of lines of code that are far more reliable than if they were hand crafted.

To sum up, a code generation approach offers the advantage of instantly generating code, which is optimized, consistent and thoroughly tested. But most importantly, it increases productivity by enabling developers to focus on the more important areas of the application, which cannot be generated by a machine.

Selecting Code for Automatic Generation

It has been noted that only a small part of application functionality can be generated by a machine (Glass, 1996). To have something generated, we must describe to the machine what exactly we expect it to generate. Describing a piece of code in a way that a machine can reproduce it, quite obviously, is more complicated than just writing it. Therefore, using this approach makes sense only if we had to write the same code multiple times. Thus, the first step in automating development is identifying recurring code patterns, which can be done through analyzing the development process decomposition of the selected type of applications.

Conceptually, data-intensive web-based applications can be described as consisting of three layers: data access, business logic and presentation. This decomposition is based on the Model-View-Controller (MVC) design pattern (Java BluePrints, n.d.), which separates the core business model (“model”) from the application logic (“control”) and the presentation (“view”). The MVC pattern decomposes the application into three distinct layers which have their specific responsibilities: data, application logic, and presentation. The data layer consists of the data storage component (in most cases – a database) and the code for accessing (i.e. storing, retrieving and modifying) the data. The application logic layer implements the business rules of the

application and is responsible for processing the data. The presentation layer is the user interface which may be a web site, a desktop application, a console application, etc.

The application business logic layer is unique to each application and does not contain any evident recurring patterns, so its code is not a good candidate for specification and automatic generation. The presentation and data layers, on the other hand, contain multiple recurring code patterns – which makes it possible for parts of this code to be abstracted and generated automatically. In order to identify the parts most suitable for automatic generation, I will look at these two layers in more detail, while consulting some of the existing research in this area.

Overview of Modeling Data-Intensive Web-Based Applications

A code generator is "a program that translates a domain specific language or specification into application source code" (Whitehead et al., 2004, p. 209). Therefore, without loss of generality, the process of code generation can be described in terms of (a) modeling the features to be generated, and (b) translating the model into code. The second part is a straightforward process and will be briefly examined in the methodology chapter of this thesis. The concept of modeling, on the other hand, is more involved and presents numerous design and implementation options, which are being actively explored by scholars and developers alike.

Most of existing systems model both, the data and presentation layers of a web-based application, decomposing the two layers into more specific conceptual parts. WebML, or the Web Modeling Language, introduced by Ceri, Fratenali, and Bongio (2000) and extended by Ceri, Fratenali, and Matera (2002) is one of the more

comprehensive solutions. This modeling framework consists of four perspectives: (1) a structural model, which is used to describe the underlying data; (2) a hypertext model, which consists of a composition model, used to specify all the ways in which data might be displayed on the page, and a navigation model, which expresses how pages are linked; (3) a presentation model, which describes the graphic appearance of pages; and (4) a personalization model, which allows the modeling of users and groups and their relations to user- or group-specific data and settings.

The AutoWeb system (Fratenali & Paolini, 2000) takes a very similar approach in implementing the HDM-Lite modeling framework, which allows the description of a web application by a schema in three parts: (1) the structure model (i.e. data model), (2) the navigation model, and (3) the presentation model. Other modeling frameworks take very similar approaches: Bochicchio and Fiore (2004) specify (1) an information design (i.e., the data model), (2) a navigation design, (3) a publishing design (i.e. presentation model) and (4) an operation design (which is a model of the data access requirements). Jacob et al. (2006a) and Jacob, Schwarz, Kaiser, and Mitschang (2006b) describe a content model (or data model), composition and navigation model, and a presentation model. Similar approaches are offered by Merialdo and Atzeni (2000), Milosavljevic et al. (2002), Zhang, Chung, and Chang (2004), Jensen, Tolstrup, and Hansen (2004) and Turau (2002), with only slight differences.

To sum up, existing code generation systems facilitate modeling of data-intensive web-based applications along the following lines: (1) the structure of underlying data, (2) data access operations, (3) web site navigation, and (4) web page presentation. These

modeling directions are easily combined into modeling the data and presentation layers, in accordance with the previously described MVC design pattern; with the data layer represented by the data model and the data access operations, and the presentation layer – by the application's web pages and the linkages between them (or the navigation system). However, in my opinion, modeling the presentation layer presents significant problems, which I will discuss in the next sections.

Argument against Modeling Web Pages

The problem of modeling web pages can be viewed in different contexts. Certainly, there are benefits to it: the development of the application becomes more systematic and, thus, the entire process can be streamlined and subjected to the rigorous approach of software engineering. Nevertheless, consider a basic content management system, which consists of web pages providing the ability to add, edit, delete and view data. Even such basic functionality requires a very fine level of detail, which deals with user interaction requirements unique to each application, which cannot be easily described in a model. Modeling these fine details is pointless: in that case, the model's level of abstraction would have to be the same as the code's; and to achieve that it would be necessary to reinvent a general purpose programming language – which makes little sense.

A simplification of the user interface through abstraction is another approach. Milosavljevic et al. (2002) propose to abstract the user interface of data-intensive web-based applications to the following three types: (1) row per page, (2) table per page, and (3) parent-child per page. This, in my opinion, is both an oversimplification and over-

specification. For example, a row-per-page is described as a page displaying a single table row (i.e. displaying one data record, which is stored in the table row), enabling the addition of a new row and the updating of an existing row. However, additions and modifications of records can take place in various contexts, and one record per page is only one of them. As a matter of fact, a record per page format is best-suited for displaying, or editing a record, but not adding or deleting one. At the same time, it is an oversimplification: a record, or a data object, can be represented by a join from multiple tables. The same reasoning can be applied to table per page and parent child per page layouts.

Therefore, I conclude that neither modeling the user interface's fine details, nor simplifying it through abstraction is a reasonable solution. In my opinion, user interface requirements are unique to almost every application and too important to be dropped.

Argument against Modeling Website Navigation

Modeling the navigation of a web application means describing the linkages between web pages and displaying this system on the web site as a set of navigation menus. There are two reasons that I disagree with this approach.

Reason #1: two different systems. Web pages and web site navigation are two different systems and should not be confused with one another. A page is not a menu option, although a menu option is always mapped to a page. Why? Because a "real world" web application may consist of hundreds or even thousands of pages – dynamic or not – all of which simply will not fit on the web site's menu. Each menu option, obviously, should be associated with a specific page. But it is not a 1-to-1 relationship,

for there may be plenty of pages mapped to the same menu item. Using web pages to model the web application's navigation system is feasible only for relatively small applications. Besides, website navigation is a tool for navigating the site's content – it's not a catalog.

Reason #2: requirement for dynamic navigation. Modeling navigation will render a web site's structure static. A web site can be viewed as a collection of content presented in the form of web pages. The way these web pages are organized and interlinked does not have to be static. Applications of dynamic navigation range from trivial content management features, enabling the site administrators to change the site's entire navigation structure (new pages are created, existing pages are moved around, menus are altered) to adaptive websites, which change their structure based on the users' preferences, or even their navigation history – the possibilities are endless. Therefore, website navigation, in my opinion, should not be modeled, but rather made as dynamic as possible.

Selecting the Data Layer for Generation

Examining existing code generation systems and current research, as well as the actual code of existing data-intensive web-based applications, I found that the data layer contains the most recurring code patterns in such applications: the data access code, as well as data access functionality in general, remains almost the same regardless of the application domain. Therefore, I concluded, that the data layer is the optimal part of the application for abstracting common features and automatic code generation.

Modeling Data and Data Access Functionality

In the previous section, I identified the data application layer as the application's most optimal part for modeling and automatic generation. In this section, I will discuss how such modeling can be implemented.

The Entity-Relationship Model Approach

The most common approach to model the application's data is to use the entity-relationship (ER) model, introduced by Chen (1976) and later extended in several ways. The basic idea of the ER model is that "using sets and relations we can model objects of the real world and their inter-relationships" (Vigna, 2002b, p. 35). Consider a basic news website: in this example, "author" and "article" might be the main entities. Each entity has a set of attributes: the author entity might have attributes like "first name" and "last name," the article entity might have attributes like "title," "body" and "date." Relationships model connections between entities. In our example, we may have an "authorship" relationship between an author and an article.

Most of the code generation systems I have examined, such as the ones described by Ceri et al. (2000), Fratenali et al. (2000), Vigna (2002a), Vigna (2002b), Vigna (2002c), Vigna (2003), Whitehead et al. (2004) and others, use the ER model to conceptualize the application's data model. The benefits of this approach are two-fold: first, this approach does not require any database-specific knowledge – the reification process (i.e. transforming a conceptual schema into a specific logical database schema) generates the database automatically; secondly, a system based on the ER model can interpret the data model in a deeper way, compared to a system where the data is

specified as database tables, since it has the knowledge of the underlying conceptual model.

On the other hand, specifying an application in terms of entities and relationships adds a level of abstraction, which causes additional implementation complexity. Besides, specifying the database structure directly provides the developer with more fine-grained control over the implementation of the data model, which, in my opinion, is crucial in real world applications.

Once the data model is specified, the next step is to describe the data access functionality. There are two options: we can either describe the data functionality explicitly – i.e. specifying each data access method; or we can have the data access methods derived from the data model alone.

Defining Data Access Operations

An approach describing all the data access functionality is offered by Jacob et al. (2006a) and Jacob et al. (2006b). These two papers propose a code generation framework where describing the data model of the application is separated from describing the data-related functionality.

According to Jacob et al. (2006a), most of the currently available modeling environments for these kinds of applications enable the specification only of simple data access functionality. However, the authors argue, web applications not only provide large amounts of data to be displayed on the web, but also require powerful operations that determine the manner of content provision and allow data manipulation (p. 77). Indeed, functionality like persistent shopping carts, reporting tools, or flexible data grids, that

enable the user to browse through thousands of records, require more than the standard create/read/update/delete data access functionality. As Jacob et al. (2006a) correctly states, the required operations not only change simple content entities of the web application, but also add and modify relations between these entities. According to Jacob et al. (2006b), today's web applications have to provide at least the functionality allowing (1) to add, alter and delete entities or relationships between them, and (2) to filter and sort entities according to some criteria.

To solve this issue, the authors propose an Operation Model – a framework which enables the modeling of data access operations. Figure 1 displays a simplified example of this model.

```
<content>
        <object name=”User”>
                <attribute name=”Name” type=”string”/>
        </object>
</content>
<operations>
        <object name=”User”>
                <add/><modify/><delete/>
                <sort attribute=”Name” direction=”ascending”>
                        <filter compare=”equal” attributes=”Name”/>
                </sort>
        </object>
</operations>
```

Figure 1. Operation Model Example.

The example in Figure 1 defines the data access operations for the “user” entity, defined in the content model. The following data access operations are defined: “add,” “modify” and “delete.” Also, whenever a collection of user objects is retrieved, it will be

sorted by the “name” attribute in ascending order, providing the option to filter the results by name.

I disagree with two issues in this example. First of all, in my opinion, it is unnecessary to specify the obvious: every data object will require the add/modify/delete operations. Secondly, when displaying a collection of records, it is helpful to provide the user with the option to sort by multiple fields, both in ascending and descending order. Therefore, the “sortability” criteria could be attached to an attribute of an object in the data model. For example, if we have a “user” object with a set of attributes, some of them – such as “name” – could be marked as “sortable,” while others – such as an attribute like “biography” do not need to be sortable.

Overall, the operations model is a good example of specifying data access functionality; I found the syntax to be intuitive and parts of the authors’ strategy to be applicable to my own code generation framework.

Deriving Data Access Operations

Another approach is to specify only the data model of an application and generate the data access code based on the data model alone. This approach is based on the assumption that the data model itself is sufficient for defining the necessary data access functionality. In my opinion, the following functionality can be derived from the data model for each object: (1) adding, modifying, reading and deleting a record, and (2) reading a collection of records based on some criteria – with a record representing an entity or a relationship.

However, existing code generation systems which use this approach generate only the very basic data access methods. For example, the code generation system described by Milosavljevic et al. (2002) offers the very basic functionality: in addition to the standard create/read/update/delete methods, it offers methods like *Get[object]Count*, *Insert[object, index]* and *checkContrsaints*. This situation is easily explained by the potential complexity of the data access functionality of even a trivial data model. Consider the following issues:

a. When a record is created or modified, some attributes are supplied by the user, while others, such as the record identity, a date and time of the record's creation and/or modification, any attributes derived from other attributes, are generated by the system.
b. When a record is deleted, in some cases other records, known as "weak entities" in the ER model, should be automatically deleted with it, for they cannot exist without their parent record (for example, orders and suborders, or books and editions). In many cases, the choice depends on numerous criteria determined by the application's business logic.
c. Retrieving a single record is relatively easy; however, retrieving a collection of records may involve selecting what fields to retrieve.
d. Retrieving a collection of records can be done based on some criteria; deriving this criteria from a data model without explicit specifications is not a trivial task.

These issues raise an important question to consider: what should a data model specify besides the structure of the data (i.e., entities and their attributes)? How detailed should a data model be to provide enough information for the system to be able to automatically derive the necessary data access functionality?

Extending the Data Model with Data Access Details

Much of the data access functionality depends on the application's business logic. Therefore, there are numerous tradeoffs in specifying the details of this functionality: specifying too much will tie the data model to the application's business logic and presentation layer, specifying too little may be too much of a simplification.

One example of such a tradeoff is the choice of a data type system. Turau [8] proposes a system which provides only one data type: a string, with all the conversions delegated to the business logic layer. I disagree with this approach, for it prevents from specifying data type-specific requirements in the data model.

Another example of such a tradeoff is specifying data validation requirements in the data model. Turau (2002) defines validation requirements for each attribute, including ranges of accepted values and multiple validation rules with basic Boolean logic. I disagree with this approach. The data model can and should define rules for structural integrity of the data, such as foreign key and unique value constraints, as well as attribute data types. However, data entry validation rules are based, to a great extent, on the application's business requirements and, therefore, belong in the business logic layer. It is conceivable that these requirements may change over time (like the set of accepted

payment methods in an e-commerce application), in which case hard-coding them into the data model will render the application too rigid.

However, one of the main considerations in specifying the data model is choosing a way to describe the requirements for displaying data in the presentation layer. The most comprehensive solution is offered by the WebML language (Ceri et al., 2000) in the form of its composition model. This model specifies content units which make up the application's web pages. This model is coupled with the presentation layer, which is an approach I am trying to avoid; however, some of those ideas, if decoupled from the presentation layer, are quite interesting.

The composition model consists of several content units, the most relevant of which are demonstrated in Figure 2.

```
<dataunit id=”SomeUser” entity=”User”>
        <include attribute=”firstName”/>
        <include attribute=”lastName”/>
</dataunit>
<dataunit id=”SomeId” entity=”User”><includeall/></dataunit>

<multidataunit id=”SomeId” entity=”User”>
        <dataunit id=”SomeUser” entity=”User”>
                <includeall/>
        </dataunit>
</multidataunit>

<indexunit id=”SomeId” entity=”User”>
        <description Key=”lastName”/>
</indexunit>

<scrollerunit id=”SomeId” entity=”User” first=”yes” last=”yes”
        prev=”yes” next=”yes”/>

<filterunit id=”SomeId” entity=”User”>
        <searchattribute name=”lastName” predicate=”like”/>
</filterunit >
```

Figure 2. WebML Composition Model Syntax.

A *Data Unit* shows information about a single object: an instance of an entity. It is defined by an entity and a selection of attributes which are included in the unit. A *Multi-Data Unit* displays a collection of data units, by repeating a single data unit. An *Index Unit* presents multiple instances of an entity as a list. A *Scroller Unit* provides commands to scroll through objects in a list and is used in conjunction with a data unit. A *Filter Unit* provides a search feature which enables the user to search (i.e. filter) through the data units displayed as a list and, thus, is used in conjunction with a *Multi-Data Unit* or an *Index Unit*.

In my opinion, there are parts in this model, which may be improved. For example, the *Index Unit* should specify the displayed attribute: an index displaying a selectable list of users might use the “last name” attribute as the one to be displayed in the list, but the “user Id” – as the key for selected records. Another example is the *Scroller Unit*, which, in my opinion, is not necessary at all: any entity must come with this functionality. Besides, a scroller (or a pager) belongs in the presentation layer and should be attached by default to any collection of data objects. Finally, a *Filter Unit* belongs in the presentation layer, with its functionality derivable from the data model.

Regardless of the mentioned issues, the WebML approach presents a detailed and clean way to specify how data should be displayed on web pages. It offers the ability to specify the fine-grained details of data retrieval operations and, if decoupled from the presentation layer, might have been a viable option for this study’s code generator. Nevertheless, I believe that specifying the data access requirements in the data model as

additional attributes should be sufficient for deriving the required data access functionality – which I intend to demonstrate in the experimental part of this thesis.

Summary of Background Material and Related Work

In this chapter, I described data-intensive web-based applications as the subject of the thesis and defined them as systems, which require comprehensive data access functionality for providing web-based “read”" or “read/write” access to data stored in a data repository, such as a database. I showed that automatic code generation is a solution to numerous implementation issues, which arise as a result of recurring functionality requirements in such applications.

I explained that only part of the application can be automatically generated. To identify this part, I used the Model-View-Controller design pattern and analyzed this type of applications by decomposing it into three conceptual layers: data, business logic and presentation. After examining existing code generation systems, current research and the code of existing data-intensive web-based applications, I concluded that the data layer is the optimal part of the application for automatic code generation.

After reviewing the concept of data modeling, I concluded that there are two general approaches to describe data access operations: explicit and implicit. The explicit approach is based on describing every data access method individually. The implicit approach is based on deriving the required data access methods from the data model alone – which is the core of my research hypothesis.

CHAPTER 3

METHODOLOGY

In this chapter, I will describe the target application architecture, the process of defining the specific code to be generated through abstracting common functionality, and the tasks I need to accomplish in order to test my hypothesis. Since my modeling approach is less abstract than the entity-relationship model and is closer to a logical database model (i.e., a model based on tables, records and fields), I will use database terminology: I will refer to data objects, or entities, as "records;" and to their attributes as "fields."

Target Application Architecture

The architecture of the target application is based on the Model/View/Controller design pattern, discussed in the previous chapter, and consists of three conceptual layers: data, business logic and presentation. The data layer consists of two parts: the data storage component and the application-level data access code, both of which will be described in this section. However, many of the data access requirements are set by the application's presentation layer – which is why I will briefly describe these requirements.

Data Access Requirements Set by the Presentation Layer

Adding, modifying and displaying a single record is a trivial example of data access requirements set by the presentation layer. However, when dealing with collections of records, such requirements become more complex. Retrieving and displaying collections of records is a requirement for all data objects; it enables a user to browse or search through a set of records and select one or more records for viewing,

editing, deleting, or manipulating in other ways. Figure 3 demonstrates a possible implementation, including the following features:

1. Sorting. For convenience, records should be sortable by all fields displayed in the list in both, ascending and descending order.
2. Filtering. The size of the displayed collection may be reduced by entering filtering, or search criteria.
3. Paging. This feature provides the user with the option to view the collection "one page at a time." This becomes absolutely necessary with large collections: it is not unheard of for a table to contain millions of records – displaying all of them on one page is impossible. The pager on Figure 3 is an example of complex functionality which can be implemented for all data objects by default – as opposed to the approach used in WebML and discussed in the previous chapter.

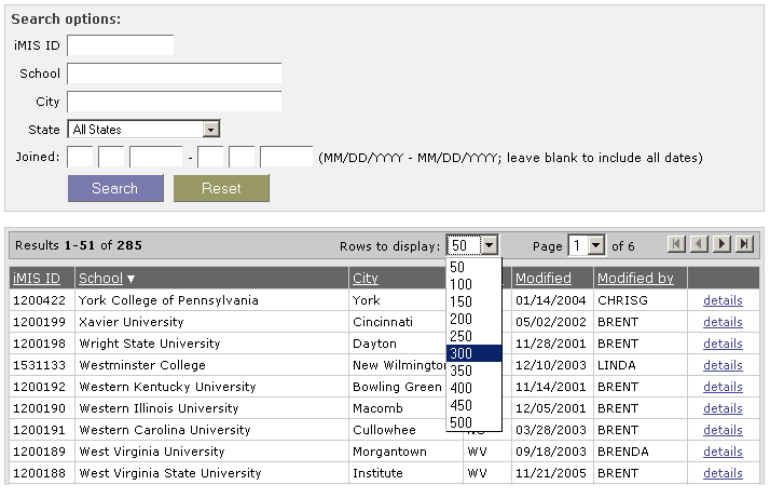

Figure 3. Displaying a Collection of Records.

The described features are usually combined and used whenever there is a need for displaying a set of records. Another example of this combination is displaying and modifying a collection based on a relationship. This is a very common requirement and it becomes essential even with a small collection of records: consider assigning multiple users to a role, or, visa-versa: multiple roles to a user. In this case, the collection includes an additional field, which indicates the existence of a relationship between each record and some record from another table.

Data Storage Component

Relational database as the data storage component. The code generation system described in this thesis assumes that the application's data storage component is a relational database. Turau (2002) argued that there is a current trend to use plain text as a format for storing knowledge persistently, quoting Hunt and Thomas (2000), who "believe that the best format for storing knowledge persistently is plain text." Nevertheless, they acknowledge that there are drawbacks to this paradigm; for example, it may be computationally more expensive to interpret and process a plain text file. I agree with this concern. According to Codd (1970), the creator of the relational model, which is the foundation of relational databases, the number of possible criteria by which the data can be manipulated is very large: "In a large data bank each subset of the data has a very large number of possible (and sensible) descriptions, even when we assume (as we do) that there is only a finite set of function subroutines to which the system has access for use in qualifying data for retrieval" (p. 382). A relational database, compared to a text file, is by far superior as a tool for storing and retrieving data, because it allows

"the designer to focus on the logical representation of the data and its relationships, rather than on the physical storage details" (Rob & Cronel, p. 58).

Database design. Discussing the design of a database's logical model – i.e. the tables which represent data objects and relationships between them and store instances of both, data objects and relationships, as records – is beyond the scope of this thesis. However, I will briefly describe the approach I used to represent data objects and relationships as database tables.

There were three abstract data concepts I needed to represent in my database design: (1) a data object, (2) a one-to-many relationship, and (3) a many-to-many relationship:

1. A data object is the same as an entity in the Entity-Relationship model and is implemented as a table, which I refer to as a *record table*. A record table contains records, each representing one data object. For convenience, each record has a unique numeric identifier, which is an automatically-incremented integer field. This identifier is used as the record's "primary key" – a field uniquely identifying a record.
2. A one-to-many relationship describes a situation in which an instance of one data object can be related to zero or more instances of another data object. This type of relationship is implemented with the help of a "foreign key" – which is a record's primary key, stored in another record – thus, relating one record to the other.
3. A many-to-many relationship describes a situation in which one data object can be related to zero or more instances of another data object and visa-versa. For this

case, I use a table, commonly known as a *link table*, in which each record consists of two foreign keys. The primary key of this record is the combination of the two foreign keys – which, as an extra bonus, guarantees that there are no duplicate relationships.

In addition to tables, a database contains stored procedures, which are database-level data access methods. There are numerous reasons for using stored procedures in combination with application-level data access code, which include better performance, ease of modification and application security. Discussing these reasons is beyond the scope of this thesis.

Application-Level Data Access Code

The application-level data access code contains classes which provide a public interface for accessing the database directly or through its stored procedures. To show why each class is necessary in this implementation, consider a simple example of writing a user's name to a web page. The following steps must be made: (1) connect to database, (2) retrieve the required record and store it in some data structure, (3) disconnect from database, (4) write the name fields of the record to the web page. An optimized implementation of this operation written in c# contains at least 15 lines of code! Obviously, such code should be written only once. Ideally, to display a user's name on a web page, I would like to write the following code:

```
User myUser = new User(Page.Request["UserId"]);
Response.Write(myUser.Name);
```

The first statement retrieves the identification of the record to be retrieved and instantiates a class which exposes the fields of a user record. The second statement writes

the user's name to the web page. Creating, updating, deleting and displaying sets of records should be just as simple.

This code simplicity can be achieved in multiple ways. My approach is based on abstracting the most common database access functionality into one abstract class, which I will refer to as the *DataClass*, which is then extended by concrete classes. In addition to the DataClass, there are several abstract classes, which serve as base classes for concrete "data containers" – i.e., wrappers for standard data structures which provide convenience features (i.e., "syntactic sugar") for manipulating collections of records. These abstract classes include the following:

1. *IDataTable*: an interface for a table of records;
2. *IDataRow*: an interface for a record;
3. *IDataField*: an interface for a record field;
4. *AbstractDataTable*: a data structure holding a collection of records, which implements the IDataTable interface.

By using the described encapsulation approach, we end up with code which is much more complex and difficult to write and maintain, yet we are protected from run-time errors caused by incorrect direct references to the structure of data in the database. In the next section, I will show how all this code can be automatically generated – which eliminates any issues which would have been caused by the added complexity. The code of the described abstract classes is available in Appendix A.

Selecting the Code to be Generated

The application's data layer contains a set of classes and database objects (tables and stored procedures), which provide data storage and data access functionality. In order to generate these items, we need to identify the parts which can be abstracted and generated based on an application's data model.

Abstracting Code for Generation

Upon closer examination, it becomes apparent that the items in the data layer – the database objects, such as the tables and the stored procedures, and the concrete application-level classes – are nearly identical for any data object. The only differences are in (a) the names of the item, (b) the parameters, and (c) the set of data access methods. Following is a detailed description of these items. The sample code for some of these objects is available in Appendix B.

Database table for each data object and each data object link. A table is generated by executing a SQL statement. The only parts which need to be specified are the name of the table and the name and data type of each parameter.

Stored procedures for each data object and each data object link. Stored procedures, like tables, are generated by executing SQL statements. Each procedure is unique in terms of its data object name and, in some cases, its parameters. Data object links require an identical set of data access methods for their creation and deletion by various criteria. However, different data objects, require unique data access methods, based on their fields and, most importantly, their relationships with other data objects.

This problem is the core of this thesis's hypothesis, which I will explore in the next chapter.

DataField and *DataRow* classes for each data object. These classes implement the *IDataField* and *IDataRow* interfaces, providing access to a record's fields and the data associated with a single record field (such as custom sort expressions, displayed titles, etc.). Only the data object name and parameters are different.

DataTable and *DataList* classes for each data object. These classes extend the *AbstractDataTable* class and provide access to each record in the table by exposing collections of *DataRow* classes and *DataField* classes. A *DataList* provides a collection of records with a minimized set of field. In addition to the data object name and parameters, data objects differ in terms of the set of fields which are displayed in these classes.

Record class for each data object. This class extends the *DataClass* and provides access to a single record, exposing its fields and the "update" method. The code differs only in terms of the data object's name and its attributes.

RecordData class for each data object. This class extends the *DataClass* and provides access to methods operating on collections of records and methods, which do not require the instantiation of a single record class. It exposes the previously described stored procedures and differs not only in terms of its name, but also in terms of the set of stored procedures it must expose.

RecordLink class for each data object link. This class extends the *DataClass* and provides access to methods facilitating the creation and deletion of data object links. The code differs only in terms of the data object names which constitute a link.

List of Code to be Generated

In conclusion, I provide a list of specific items to be generated for the data layer of an application.

For each data object:

1. a database table;
2. a set of stored procedures facilitating data access;
3. a *DataField* class for each data object attribute;
4. a *DataRow*, *DataTable*, *DataList*, *Record* and *RecordData* class.

For each data object link:

1. a database table;
2. a set of stored procedures facilitating data access;
3. a *RecordLink* class.

Experiment Design

This section describes the specific tasks I need to accomplish in order to test my hypothesis. These tasks can be divided into (a) implementation tasks, which deal with describing the data model, defining rules for data access methods and constructing the code generator; and (b) testing tasks, which deal with applying the implemented system to generating the code for real applications and measuring the results of using this approach.

Implementation Tasks

Describing the data. The approach I am describing in this study is to generate code based on the data model of the application. The data model has to be expressed in a way, which can be understood by the code generator. The data model must include data objects and their attributes, and provide a way to specify relationships between data objects, as well as additional information which will define the details of data access operations. Designing a data definition language for such a model is the first implementation task.

Deriving data access functionality. The previous section included a list of code items – both database-level and application-level – which need to be generated. However, some of these items mentioned sets of data access operations without enumerating them explicitly – which is the main point of this study's hypothesis. Therefore, I must come up with rules which define the kinds of operations which will be generated for different data model combinations. Defining these rules is the second implementation task.

Implementing the Code Generator. Building the actual code generator, which will take as input a data model, described in the data definition language I design, and will generate the required code is the third implementation task.

Testing Tasks

Testing approach on real applications. The first step in testing the hypothesis will be checking the feasibility of the developed system on real applications. For this, I will attempt to generate the data layer for three data-intensive web-based applications. Each

of these applications represents a distinct set of requirements, so the combination of the three will present a diverse scope of development requirements.

The first application, referred to as Witness Identification, is a web-based tool used in criminology, specifically – for eyewitness identification. A user (a witness) is presented with a sequence of head shots of suspects, selected from a set of several hundred thousand images. The application is relatively simple; its main challenge is manipulation of a very large set of data.

The second application, referred to as Account Reporting, is a more complex system – an online tool which provides university's constituents with access to various university accounts. The system uses multiple databases and requires elaborate data access functionality to generate complex data reports.

The last application, referred to as PRSSA, is a collection of web sites with a complex content management system, which includes regular web sites, a blog, a career web site and numerous administrative functionality. The challenge presented by this system is the amount of different features it contains.

Measuring results. To determine how successful was my approach, I will conduct the following measurements for each of the described applications:

1. Amount of generated code. The following measurements will demonstrate the amount of the generated code in regards to the scope of each application. For each application, I will measure the following:

- number of data objects, data object links and data object attributes;
- number of generated database tables;

- number of generated database stored procedures and lines of code;
- number of generated classes and lines of code;
- total generated lines of code.

2. Effectiveness of approach. This measurement will demonstrate what part of the application's data access code was generated. For each application I will measure the following:

- lines of code: total, generated, percentage of generated;
- stored procedures: total, generated, percentage of generated.

3. Efficiency of Approach. This measurement will demonstrate what part of the generated data access code was actually used in the application. For each application I will measure the following:

- stored procedures: generated, generated and used in application, percentage of used.

I consider a stored procedure to be an indicator of a data access operation, as opposed to a public method, because most of the public methods provide wrapper/convenience functionality and, therefore, are not essential for the application's functionality. Stored procedures, on the other hand, are the actual link between the application code and the data stored in the database; they may be also described as corresponding to the primary data access methods, whereas all other methods are secondary and use the primary methods for accessing the data. By "used" I imply that the stored procedure were accessed from the application's business or presentation layer, or from another part of the

data layer, which, in its turn, was directly or indirectly accessed from the application's business or presentation layer.

CHAPTER 4

RESULTS

In this chapter, I will describe the results of carrying out the implementation tasks, dealing with describing the data model, defining rules for data access methods and constructing the code generator; and testing tasks, dealing with applying the implemented system to generating the code for real applications and measuring the results of using this approach.

Data Model Specification

Introduction to Basic XML Syntax

The data definition language I designed uses XML syntax and serves for describing the application's data model, including some additional features which allow the specification of data access details. The application is described in one file, referred to as the schema, where the application's data model is represented as a tree. To describe the structure of this tree, I will use XML tree concepts, such as element, child and parent elements, and element attributes. Figure 4 contains an example of such a tree:

```
<ElementA attribute1=”abc” attribute2=”xyz”>
        <ElementB/>
        <ElementC/>
</ElementA>
```

Figure 4. XML Tree Syntax.

There are three elements in this tree: *ElementA*, *ElementB*, *ElementC*. *ElementA* has two children: *ElementB* and *ElementC*, both of which have *ElementA* as their parent

element. *ElementA* also has two attributes: *attribute1* and *attrbute2*, whose respective values are “abc” and “xyz.” For brevity, I will refer to these concepts as element, parent, child, and attribute.

Structure of the Schema

The structure of an application schema is displayed in Figure 5. The indents signify parent-child relationships between elements. The text in parenthesis specifies the number of the element’s occurrences. If the element has no children, it has a text value, which is a string, or is specified by a list of allowed values. The complete description of this data definition language is available in Appendix C in XML Schema format.

The tree’s root element is the application element, which contains one or more namespace elements. A namespace element represents a group of data objects: a single application may have more than one namespaces. Each namespace node contains one name element, which is the namespace’s name, and one or more *Class* elements. Starting with the *Class* element’s tree level, the schema becomes rather complicated, which is why I will provide a more detailed description of those elements.

Class element and its children. The *Class* element represents a data object. Each *Class* element contains one *Name* element, one *Type* element, one *Table* element, and zero or more *AdditionalSproc* elements. The current code generator supports classes of four types:

- a *record* class represents a data object with all standard data operations allowed;
- a *readonly* class is a data object which cannot be created, modified or deleted;

```
Application (1)
   Namespace (1 or more)
      Name (1)
      Class (1 or more)
         Name (1)
         Type (1): record, link, readonly, final
         Table (1)
            Name (1)
            External (0 or 1): true, false
            Field (1 or more)
                Name (1)
                SqlDataType (1)
                Identity (1): true, false
                PrimaryKey (1): true, false
                ForeignKey (0 or more)
                    RefTable (1)
                    RefField (1)
                Unique (0 or 1): true, false
                Encrypted (0 or 1): true, false
                Display (0 or 1)
                ExcludeFromTable (0 or 1): true, false
                IncludeWithParentTable (0 or 1): true, false
                IncludeInList (0 or 1): true, false
                DefaultSort (0 or 1): true, false
                ReadonlyType (0 or 1): created, modified, timestamp
            AdditionalField
                Name (1)
                SqlDataType (1)
                Sql (1)
                SortExpression (1)
                Display (0 or 1)
                ExcludeFromTable (0 or 1): true, false
                IncludeWithParentTable (0 or 1): true, false
                IncludeInList (0 or 1): true, false
                DefaultSort (0 or 1): true, false
         AdditionalSproc (0 or more)
             Name (1)
             Param (0 or more)
                 Name (1)
                 CsDataType (1)
                 Encrypted (0 or 1): true, false
             ReturnType (1)
             ReturnField (0 or more)
                 Name (0 or 1)
                 CsDataType (0 or 1)
                 SortExpression (0 or 1)
                 Display (0 or 1)
```

Figure 5. Application Schema Structure.

- a *final* class is a data object which cannot be modified after creation. For example, a data object dealing with activity logs, placed orders, etc;
- a *link* class is a link between two data objects.

Table element. The *Table* element holds the collection of the data object's attributes (*Field* elements). It also contains additional information about the table, such as

whether the table is external or not. External tables are declared, but no fields can be specified: these tables represent data from external data sources (a common situation in a "real world" application), so they are ignored by the code generator. Nevertheless, they must be defined, since generated data access methods might refer to them.

Field element and its children. This element represents an attribute of a data object. The element's children specify various information, used by the code generator. They also specify additional information and criteria used by the application's presentation layer to display the field's data.

AdditionalField element and its children. This element represents an attribute added to a data object during data retrieval operations. For example, retrieving records from a "user" table might require constructing an additional "full name" field on the fly by combining the "first name" and "last name" fields. Like with a regular field element, the element's children specify additional information and criteria used by the data access code and the application's presentation layer to retrieve and display this data.

AdditionalSproc element and its children. These elements represent custom data access methods, which must be manually defined. The stored procedures are created manually; their definition in the schema signals to the code generator to generate methods exposing these custom stored procedures to the application's business logic layer.

Consider the following basic example: we have a blog, which consists of posts; a post can be made by any user; a user can post, modify, or delete blog posts based on their permissions, which are hard-coded into the system. Besides, a user may have multiple permissions. Figure 6 contains part of the schema for such an application.

```
<class>
 <name>User</name>
 <type>record</type>
</class>
<class>
 <name>Permission</name>
 <type>readonly</type>
</class>
<class>
  <name>UserPermission</name>
  <type>link</type>
  <table>
   <field>
     <name>UserId</name>
     <primarykey>true</primarykey>
     <sqldatatype>int</sqldatatype>
     <foreignkey>
       <reftable>User</reftable>
       <reffield>Id</reffield>
      </foreignkey>
    </field>
    <field>
     <name>PermissionId</name>
     <primarykey>true</primarykey>
     <sqldatatype>int</sqldatatype>
     <foreignkey>
       <reftable>Permission</reftable>
       <reffield>Id</reffield>
      </foreignkey>
    </field>
  </table>
</class>
```

Figure 6. Sample Application Schema.

This example demonstrates how data objects and relationships between them are described in this data definition language. The full version of this basic example (although the non-essential parts are not included) is available in Appendix D.

The described data definition language is enough for making a list of the required data access methods and generating the code for the data layer – which will be discussed in the next two sections.

Defining Common Data Access Methods

Defining the rules for generating data access methods is the underlying concept in this study. Instead of specifying each method explicitly, I provide rules for generating the

most commonly used methods automatically, based on the data model alone. The methods can be grouped into the following types: (1) adding a record, (2) modifying a record, (3) reading a record, (4) deleting a record, and (5) reading a collection of records based on some criteria. In all of these cases, a record may represent an entity (a data object) or a relationship (a data object link).

There were four main problems I had to solve to implement this solution:

1. dealing with attributes which are generated or updated automatically;
2. dealing with weak entities;
3. specifying different sets of fields when retrieving collections of records.
4. defining the set of data access methods.

I solved the first problem by adding the “readonly type” child element to the field element in my data definition language. Fields marked as “readonly” are treated in a special way, based on the “type” attribute of this element. For example, if a field is marked “created,” its value is generated automatically only once, if it is marked “modified,” its value is generated each time the record is modified.

I solved the second problem by adding a required parameter to the delete method, which determines whether all child entities should be deleted as well. This is a partial solution though, for only part of the child entities might be weak entities, whereas others might be strong entities, which should be preserved even after their parent is deleted. For example, deleting an order record, containing data on the customer, payment and delivery information, should automatically remove all the order-related records, such as sub-orders, containing data on each ordered item (like price and quantity). However, deleting

this order record should not delete the customer referenced in it. Therefore, the solution to complex cases is to overwrite the delete method's default implementation.

The third problem – determining what fields to include in the set when retrieving collections of records – was solved in a way similar to the first problem's solution. Instead of designing a modeling approach like WebML's Composition Model (Ceri et al., 2000), I added several additional child elements to the *Field* element in my data definition language:

- The *ExcludeFromTable* element specified whether a field was to be retrieved in a collection of records or not (the default value being "false"). For example, marking this element as "true" for a large text field ensures that the field is never included when a collection of records is retrieved.
- The *IncludeWithParentTable* element specified whether a field was to be included in a "join" operation, when two or more tables are joined (the default value being "false"). For example, if table *A* has a relationship with table *B*, retrieving a record from *A* will include all fields from *B*, which have this element marked as "true." Consider retrieving a "user" record, which has a field "role Id," which is a reference to the "role" table, with the field "name" marked as "IncludeWithParentTable." In that case, the retrieved user record will always include the name of the user's role.
- The *IncludeInList* element specifies whether the field is included when the minimum set of fields is retrieved (the default value being "false"). For example, displaying a list of records in a drop-down box requires only the key

field and the field to be displayed. A “user” record might include many fields, but marking only the “name” field as “IncludeInList” (the primary key is retrieved by default), the retrieved collection would include only the unique identifier and the name – which is exactly what is needed.

In order to design the rules for generating data access methods, I decomposed the data access functionality into the following logical groups:

- Instance-related data object functionality. That includes retrieving or updating a record – i.e., methods accessing the fields of a single record.
- Non-instance-related data object functionality. This group includes creating and deleting a record and retrieving records with two different sets of fields (all-inclusive – *GetRecords* and minimized – *GetList*) based on criteria related to the record’s fields.
- Non-instance-related data object functionality for each one-to-many relationship. Same as the previous group, except the criteria is the related record – therefore, a set of these methods must be created for each relationship of this type.
- Non-instance-related data object functionality for each many-to-many relationship. Same as above, except the relation type is different, which results in a different implementation.
- Non-instance-related functionality for each many-to-many relationship. These methods do not belong to any data object: they provide functionality required by data object links.

The complete list of these methods is provided in Appendix E. Certainly, this is not an exhaustive listing of all possible data access operations; however, my assumption is that these methods are the most commonly used.

Code Generator Implementation

The code generator used in this study was implemented in c# on the .Net platform. It generates SQL code for the database-level part of the code, and c# or VB.Net for the application-level code.

There are two main approaches to code generation, often referred to as passive and active. The passive approach implies generating code only once (or re-generating it each time a modification is required). The active approach includes the option to automatically update previously generated and manually edited code. My code generator is a combination of both approaches.

The application-level code is generated using the passive approach: the generator produces a set of classes, which contain the default data access methods. If a new method is required, or an existing method must be altered, it is done by creating a subclass, which inherits all the functionality of the generated parent class, and may add methods or override existing methods. This has proved to be a viable solution. Consider the following example: the *Foo* class represents the *Foo* data object and contains methods *getA* and *getB*. We need to add a custom method *validateFoo* – which we add to *Foo*. In the next development iteration, we add a new relationship to our data model, which results in new data access functionality – in particular, the *Foo* class must now contain a *getC* method. Either we regenerate Foo – and lose *validateFoo*, or we have to find and

manually edit all the occurrences of the *Foo* data object. Both solutions are unacceptable. Instead, we create a *FooBar* class, which extends *Foo*, and add the *validateFoo* method to the new class. In this case, we can regenerate Foo anytime, with all our changes to *FooBar* remaining in tact.

The database-level code is generated using both approaches. The stored procedures follow the a pattern similar to the application-level code: there are automatically generated procedures, which are re-generated each time the generator executes, and there are custom procedures, which are not affected by the generator. However, the tables and their structure are automatically updated.

The code generator accepts as input a file with the description of the application and processes it in the following steps:

1. A *Parser* object is responsible for parsing the input and generating an parse tree. The parser is also responsible for validating the syntax and structural integrity of the schema in the input file. The objects constituting the application's abstract syntax contain detailed validation rules for each part of the application, such as checking that field lengths do not exceed their maximum values, that the data types are database-compatible, etc.
2. A *SchemaValidator* object is responsible for checking the application schema as a whole, which includes guarding against duplicate class names, duplicate primary keys, maintaining correct references in foreign key descriptors, etc.
3. A *SchemaDatabaseLoader* object creates a *Database* object based on the schema file – which is an abstract model of the database part of the

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

4. A *DatabaseHelper* object takes the *DatabaseComparer* as input, generates all the database-level code, connects to the database and updates it based on the data provided by the *DatabaseComparer*.
5. An *ApplicationLoader* object takes the parse tree as input and creates an abstract syntax tree, which is an abstract model of the application. This object is passed on to several objects, which generate the actual code.

Measuring Results

The data access code for all three applications, described in the previous chapter, was generated successfully. Appendix F contains the schema for the Witness Identification application. The two other application schemas are not included due to their size.

Table 1 displays the scope of each application and the amount of generated code.

Table 1. *Amount of Generated Code.*

	Witness Identification	Account Reporting	PRSSA
Data objects	16	14	48
Data object links	4	8	11
Data object attributes	92	97	449
Generated database tables	20	22	59
Generated stored procedures / lines of code	251 / 8,372	315 / 8,899	577 / 26,072
Generated classes / lines of code	70 / 9,956	116 / 17,181	204 / 34,397
Total generated lines of code	18,328	26,080	60,469
Total lines of code	28,767	49,232	143,690
Percentage of generated code	64%	53%	42%

To estimate the effectiveness of my code generation approach, I measured what part of the application's data access code was generated, which is demonstrated in Table 2.

Table 2. *Effectiveness of Code Generation Approach.*

	Witness Identification	Account Reporting	PRSSA
Lines of code: (stored procedures + code in files)			
- total	18,410	30,710	63,228
- generated	18,328	26,080	60,469
- percentage of generated	99%	85%	96%
Stored procedures:			
- total	257	375	658
- generated	251	315	577
- percentage of generated	98%	84%	88%

To estimate the efficiency of my code generation approach, I measured what part of the generated data access code was used in the application, which is demonstrated in Table 3.

Table 3. *Efficiency of Code Generation Approach.*

	Witness Identification	Account Reporting	PRSSA
Stored procedures: - generated - generated and used in application - percentage of used	 251 88 35%	 315 63 20%	 577 179 31%
Non-data access lines of code	10,357	18,522	80,462
Used data access lines of code (approximate values)	6,414	5,216	18,745
Total used lines of code	16,771	23,738	99,207
Percentage of data access code (used code only)	38%	22%	19%
Not used lines of code (approximate values)	11,914	20,864	41,724
Percentage of not used code	41%	42%	29%

CHAPTER 5

DISCUSSION

This study's hypothesis stated that it is possible to build a code generator which will significantly improve development of data-intensive web-based applications by generating at least 50% of the data access code based on a specification of the application's data model. To test this hypothesis, I designed an experiment, consisting of the following steps:

1) creating an XML-based data definition language to describe the target applications;
2) defining rules for generating data access methods based on the application's data model;
3) implementing the code generator;
4) using the code generator to generate the data access code for three "real world" applications;
5) measuring the results in terms of the amount of generated code, as well as the effectiveness and the efficiency of this approach.

All of these steps were successfully accomplished. In the following sections, I will discuss the major findings from this experiment, as well as some of the lessons I have learned from using this code generation approach.

Major Findings

The approach was tested on three "real world" applications with varying degrees of complexity. The results showing the amount of generated code serve as a good

example of the general complexity of data-intensive web-based applications: approximately 1,042 lines of code were generated for each database table. These numbers confirm that there is a need for tools which will simplify development of such applications, and automatic code generation may be one such approach.

Effectiveness of Approach

The results of measuring the effectiveness of this code generation approach proved to be successful. In terms of the number of lines of code, 85% - 99% of the data access code has been generated automatically. In terms of the number of stored procedures, which may be also associated with the primary data access methods, 84% - 98% of the required procedures have been generated automatically. The Witness Identification application required very little customization in terms of data access functionality, so its data layer was almost entirely generated automatically, The Account Reporting application, on the other hand, due to the added complexity of the business logic of a financial system, had numerous customization requirements; therefore, only 84% - 85% of its data layer was generated automatically.

I conclude, that 41% of the Witness Identification application (approximately 12,000 lines of code), 42% of the Account Reporting application (approximately 21,000 lines of code), and 29% of the PRSSA application (approximately 42,000 lines of code) represent unnecessary complexity, which may be considered not very efficient.

Concerns About Efficiency

Despite the hypothesis being supported, the results of measuring the efficiency of this code generation approach raise some big questions. The PRSSA application and the

Witness Identification application used only 31% and 35% of the generated primary data access methods. The Account Reporting application with a large amount of manually implemented code used as few as 20% of those methods. I did not calculate these values in terms of lines of code, because it was unnecessary: it is safe to assume that if 80% of the generated stored procedures were not required by the application, approximately 80% of the code was not required either (even though most of the code is encapsulated in private methods, this code supports the public methods, with only a small part of it supporting the entire application).

I conclude, that approximately 12,000 lines of code in the Witness Identification application, 21,000 lines in the Account Reporting application, and 42,000 lines in the PRSSA application represent nothing more but unnecessary complexity. These numbers represent anything but efficiency.

Observed Patterns

A detailed look at the generated methods and their usage revealed several patterns:

1. Methods dealing with a singe data object, such as creating, retrieving, modifying and deleting a record, are almost always used by the application when generated.
2. Only half of the generated methods dealing with data object links are used by any application: a link needs to be created, deleted, or generated for all values of one of the linking data objects.

3. When a collection of data objects is retrieved with paging criteria, retrieving that collection without paging criteria is only necessary as a list (i.e., a minimized set of fields).

However, these patterns may be used for only minor improvements in efficiency. The main problem lies in generating methods for retrieving collections of records based on relationships. There is no apparent pattern in the usage of these methods, and their numbers are overwhelming. In fact, one single data object, if having relationships with a few other data objects, may cause the generator to produce dozens of unnecessary methods (for example, the *User* object in the Account Reporting application, being related to four other data objects, resulted in 90 data access methods, only 19 of which were used.

Support of Hypothesis

Therefore, I conclude that the study's hypothesis is supported. The presented approach, indeed, generates more than 50% of the required data access code automatically based on the application's data model alone. However, while considerably improving development of such applications, this approach complicates the process, to a certain extent, by cluttering the application with unused code.

Lessons Learned

I have learned numerous lessons through conducting this experiment, most of which are either too specific in terms of implementation details, or represent tangent considerations. Nevertheless, I found some of these observations to be intriguing and relevant to the subject of this research.

Simplicity Versus Flexibility

The main thing I have learned from this experiment is that a code generation system, as well as, arguably, almost any software, is a trade-off between a flexible, yet complex system which allows the specification of numerous detailed criteria – and a rigid, yet simple system, which, in comparison, has most of the options hard-coded and generates code which is more standardized. It might appear that a flexible system is more desirable, however, my experiment proved to me that "keeping it simple" is, in fact, a better approach.

Argument for Simplicity. Take, for example, the problem of weak entities, which I briefly mentioned in Chapter 2. Vigna (2002c) suggests that to handle weak entities, there must be an identifying function from one entity to another, specifying the parent and child entities. Deletion of a parent entity implies deletion of all its children. However, I did not provide a way to specify weak entities in my schema. Instead, I added a parameter to the standard "delete" method my generator created which specified whether the system would delete all child entities upon deleting the current entity. In this case a simplified approach made the system more rigid, yet less complicated. In my opinion, in this case such a simplification was justified. Nevertheless, I must note that when the data model becomes more complex, my solution fails when dealing with "recursive weakness" – i.e., several levels of weak entities.

In general, the main drawback of a complex system is that as we move towards more complexity the data definition language's abstraction level moves towards that of a general purpose programming language. And while it is tempting to provide the ability to

specify in the data model as much as possible, we must remember that this is only a model and it is meant to be a simplification of the actual thing – because simplification is the only reason we are using it; alternatively we could simply manually implement the entire application, using an existing general purpose programming language.

Possibilities for Improvement

There are many possibilities for improving the code generation system I designed. In this section, I will briefly elaborate on some of these possibilities.

Better Data Definition Language Syntax. In regards to usability, or the convenience of using the system, the main issue I observed had to deal with the syntax of the data definition language. There were numerous issues that could be improved, yet the main issue was readability. The data model description is viewed from a monitor, therefore is should be as compact as possible. When designing my syntax, I decided to use XML elements instead of attributes to describe record fields, which dramatically increased the size of the specification and made it hard to read: a definition of a class could span multiple screens. The usage of attributes would have significantly improved this. Consider the example in Figure 7.

Dealing with Derived Fields. Some record fields have values which are derived from other fields or are calculated by the database. For example, a column like “total price” in a shopping cart application might be derived from other columns, such as price/tax/shipping, etc… Another example would be a modification field, storing the date and time when the record was last modified. This field can be updated by the database each time the record is updated.

Current syntax:

```
<class>
  <name>User</name>
  <type>record</type>
  <table>
    <field>
      <name>Id</name>
      <datatype>int</datatype>
      <identity>true</identity>
      <primarykey>true</primarykey>
    </field>
  </table>
</class>
```

Improved Syntax:

```
<class name="User" type="record">
  <table>
    <field name="Id" datatype="Id" identity="true" primarykey="true"/>
  </table>
</class>
```

Figure 7. Better Data Definition Language Syntax.

Specifying these kind of fields in the model proved to be a challenge. I came up with several types of readonly fields – such as *created* and *modified* – and specified such a field as a readonly field. However, a more general and, therefore, better solution would be to specify a value of a field. If a value is not specified, the field is treated normally. If a value is specified – the field is derived based on the specified value. The value could be an SQL statement (for example, "firstName + ' ' + lastName" for a full name), or a global function like "getdate()".

Defining Data Views for Retrieval Methods. Finally, the most important improvement could be changing the way collections of data are specified, as well as the rules for their retrieval.

It has been shown that there are numerous ways in which collections of records can be retrieved. Assuming that paging, filtering and sorting functionality can be applied to each of these collections, their retrieval methods may differ in two respects: (1) the criteria used to retrieve the data, and (2) the set of fields retrieved. Chapter 4 has demonstrated that generating an exhaustive set of data retrieval methods is very inefficient. Handling the requirements in regards to different sets of fields included in the retrieved collections has been handled with the help of the *IncludeInList*, *IncludeWithParentTable* and *ExcludeFromTable* schema attributes, which proved inadequate, as the exceptions to these rules kept multiplying as the requirements became more detailed.

An alternative solution was offered in the WebML framework, which I described in Chapter 2. WebML offered the concept of *content units*, used to model the "composition" of the application's data – i.e., how it is presented to the user. After conducting my experiment, I am convinced that the WebML (Ceri et al., 2000) approach is superior to mine: explicitly defining content units for each data object might simplify development by making the code much cleaner.

However, instead of using the content units, proposed in WebML, I would use a simplified concept, which may be described as a *data view*. Similar to WebML's *data unit*, a data view might represent a particular "view" of a collection of data objects. For example, a *User* data object might have a minimized data view for displaying in drop-down lists, an all-inclusive data view, a data view with or without specific data fields, etc... Such a data view would include a references to a data object and a data retrieval

operation, an ordered set of retrieved (or derived) data fields, as well as filtering and paging criteria.

An additional benefit of such an approach would be separation of concerns. In my approach, I have to use the data model to specify both – the data itself and how it is structured, and the details of how that data is displayed. The data view approach would eliminate this problem: views would reference data objects and their attributes from the data model and would specify how they should be displayed for the user. Besides, these specifications could be unique for each view.

Furthermore, assigning such data views to explicitly defined data retrieval methods could, potentially, solve the efficiency problem of the approach used in this research. Therefore, implementing this new solution, in my opinion, should be the first step in further research.

CHAPTER 6

CONCLUSION

Summary of the Study

Subject of the Study and Hypothesis

In this thesis I proposed a new code generation approach to developing data-intensive web-based applications. After examining the concept of specifying and automatically generating part of the application's code, I concluded that the application's data layer is the optimal candidate for abstraction, specification and automatic generation.

I hypothesized that it was possible to build a code generator which would significantly improve development of data-intensive web-based applications by generating at least 50% of the data access code based on a specification of the application's data model. My main argument was that the application's data model is sufficient for deriving most of the data access functionality. To test this hypothesis I built a code generator and tested it on several applications.

Methodology Overview

The methodology of this study consisted of several parts. First I studied the target application architecture to determine the most common data access requirements. My next step was to analyze the data access code to determine what parts of it can be generated, which included both – database-level code, such as the database tables and stored procedures, and application-level code, which served as a bridge between the database and the application's business logic and presentation layers.

My final step was to design the experiment, which included setting implementation tasks, which dealt with describing the data model, defining rules for data access methods and constructing the code generator; and testing tasks, which dealt with applying the implemented system to generating the code for real applications and measuring the results of using this approach.

Findings

After designing a data definition language, coming up with a set of rules for deriving data access operations from the application's data model, and implementing the code generator, I tested my approach on three "real world" applications, which differed in their degree of complexity.

The results of testing my approach were two-fold. On the one hand, the approach proved to be effective in the sense that 84% - 99% of the data access code was generated automatically – which supported the study's hypothesis. On the other hand, only 20% - 35% of the generated code was actually used by the application.

The Balance Between Simplicity and Flexibility

The main thing I have learned from this experiment is that a code generation system is a trade-off between a flexible, yet complex system and a rigid, yet simple system.

I took the complex route, building a system which generated a wide array of data access methods, deriving them from the application's data model. As a result, the generated code contained most of the required data access functionality – which proved to be a significant improvement in the development process. However, a significant part

of that code (65% - 80%) remained unused, making the application's code harder to understand and navigate. On the other hand, the issues of unused generated code is commonly accepted and is treated as necessary "boilerplate" code. And yet, the mere amount of unused code, as well as the added complexity of describing data retrieval rules in the data model, suggests looking for a better approach.

An optimal solution would offer the same concept of generating most of the data access methods (84% - 99%, according to the results of this study) without explicitly describing them, while preserving a balance between the simplicity of the generated data layer and the flexibility of the modeling approach. This balance might be achieved by a simple change to the modeling approach, described in the previous chapter: the introduction of data views.

Defining a set of data views for each data object might solve two major problems. First of all, that would provide a clean and simple way of describing how data is to be presented to the user – which is an important improvement in terms of separating the description of the structure of the application's data from the way this data is displayed in different contexts. Secondly, a data view has the potential to solve the main issue revealed by this study: by assigning data views to a data object, we explicitly specify what *types* of data retrieval operations are required for each object – thus, ensuring that only the *required* data retrieval methods are generated. Considering that data retrieval methods are the primary source of unused code (as opposed to data access methods in general), limiting these methods to only those which are required by the application, could, potentially, solve the efficiency problem.

Possibilities for Further Research

There are multiple possibilities for future research in this area, some of which I have mentioned in the last section of Chapter 5. The main goal of future research would be to identify ways to make this model of code generation more efficient. In this respect, an interesting direction of research would be to examine existing applications and try to identify patterns in their usage of data retrieval methods. If strong patterns are identified, that may help designing a more efficient way for deriving data access methods from the application's data model.

Other improvements to the code generation system may include generating intermediate code, as well as the use of code templates. The combination of these approaches would make the system very flexible on two levels: (1) code templates would offer the benefit of changing the target application's architecture without modifying the code generator; (2) an intermediate code representation would provide the mechanism to generate the final code in different implementation languages and, as in the previous example – without modifying the code generator.

In conclusion, I acknowledge that, although the study's hypothesis was supported, the experiment proved the suggested code generation solution to be not very efficient due to the considerable amount of unused code. Nevertheless, I maintain that the suggested approach has potential, which may be fully realized through finding the optimal balance between simplicity and flexibility, which may be discovered through further investigation.

REFERENCES

Bochicchio, M., & Fiore, N. (2004). *WARP: Web application rapid prototyping.* Proceedings of the 2004 ACM Symposium on Applied Computing. (pp. 1670-1676). Nicosia, Cyprus.

Ceri, S., Fratenali P., & Bongio, A. (2000). Web Modeling Language (WebML): a modeling language for designing web sites. *Computer Networks: The International Journal of Computer and Telecommunications Networking. 33(1),* 137-157.

Ceri, S., Fratenali P., & Matera, M. (2002). Conceptual modeling of data-intensive web applications. *IEEE Internet Computing. 6(4),* 20-30.

Chen, P. (1976). The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems. 1(1),* 9-36.

Cleaveland, C. (n.d.). *Program Generators with XML and Java*. Retrieved April 14, 2006 at http://www.craigc.com/pg/chap1.html

Codd, E. (1970). A relational model of data for large shared data banks. *Communications of the ACM. 13(6),* 377-387.

Fratenali, P., & Paolini, P. (2000). Model-driven development of web applications: the Autoweb system. *ACM Transactions on Information Systems. 18(4),* 323-382.

Glass, R. (1996). Some thoughts on automatic code generation. *ACM SIGMIS Database. 27(2),* 16-18.

Hunt, A., & Thomas, D. (2000). *The pragmatic programmer.* New York, NY: Addison-Wesley.

Jacob, M., Schwarz, H., Kaiser, F., & Mitschang, B. (2006a). *Modeling and generating application logic for data-intensive web applications.* Proceedings of the Sixth International Conference on Web Engineering. (pp. 77-84). Palo Alto, CA, USA.

Jacob, M., Schwarz, H., Kaiser, F., & Mitschang, B. (2006b). *Towards an operation model for generated web applications.* Workshop Proceedings of the Sixth International Conference on Web Engineering. Palo Alto, CA, USA.

Java BluePrints (n.d.). *Model-View-Controller design pattern*. Retrieved April 12, 2006 at http://java.sun.com/blueprints/patterns/MVC-detailed.html

Jensen T., Tolstrup T., & Hansen, M. (2004). *Generating web-based systems from specifications*. Proceedings of the 2004 ACM Symposium on Applied Computing. (pp. 1647-1653). Nicosia, Cyprus.

Merialdo P., & Atzeni P. (2000). Design and development of data-intensive web sites: The Araneus approach. *ACM Transactions on Internet Technology. 3(1),* 49-92.

Milosavljevic, B., Vidakovic, M., & Konjovic, Z. (2002). *Automatic code generation for database-oriented web applications.* Proceedings of the Inaugural Conference on the Principles and Practice of Programming. (pp. 59-64). Dublin, Ireland.

Rob, P., & Cronel, C. (2002). *Database systems*. (6th ed.). Boston, MA: Thompson Learning.

Turau, V. (2002). *A framework for automatic generation of web-based data entry applications based on XML*. Proceedings of the 2002 ACM Symposium on Applied Computing. (pp. 1121-1126). Madrid, Spain.

Vigna, S. (2002a). *ERW: Entities and relationships on the web*. Poster Proceedings of the 11th International World Wide Web Conference, Honolulu, USA.

Vigna, S. (2002b). *Multirelational semantics for extended entity-relationship schemata with applications.* Proceedings of the 21st International Conference on Conceptual Modeling. (pp. 35-49). London, UK.

Vigna, S. (2002c). *Reachability problems in entity-relationship schema instances.* Proceedings of the 23rd International Conference on Conceptual Modeling. (pp. 96-109). Toronto, Ontario, Canada.

Vigna, S. (2003). *Automatic generation of content management systems from EER-based specifications.* Proceedings of the 18th IEEE International Conference on Automated Software Engineering. (pp. 259-262). Montréal, Québec, Canada.

Whitehead, J., Ge, G., & Pan, K. (2004). *Automatic generation of hypertext system repositories: a model driven approach.* Proceedings of the 15th ACM Conference on Hypertext and Hypermedia. (pp. 205-214). Santa Cruz, CA, USA

Zhang, J., Chung, J., & Chang, C. (2004). *Towards increasing web application productivity*. Proceedings of the 2004 ACM Symposium on Applied Computing. (pp. 1677-1681). Nicosia, Cyprus.

APPENDIX A

ABSTRACT DATA ACCESS CLASSES

```
using System;
using System.Collections;

namespace Prssa.Core
{
	public interface IDataTable : IEnumerable
	{
		int TotalRowCount { get; }
		int RowCount { get; }
		int FieldCount { get; }
		ArrayList Rows { get; }
	}
}
```

Figure A1. IDataTable.

```
using System;
using System.Collections;

namespace Prssa.Core
{
	public interface IDataRow
	{
		int Id { get; }
		bool Selected { get; }
	}
}
```

Figure A2. IDataRow.

```
using System;

namespace Prssa.Core
{
	public interface IDataField
	{
		string DataField { get; }
		string SortExpression { get; }

		string Display { get; }
	}
}
```

Figure A3. IDataField.

```
using System;
using System.Collections;

namespace Prssa.Core
{
	public abstract class AbstractDataTable : IDataTable
	{
		public AbstractDataTable(ArrayList rows, int totalCount)
		{
			this.rows = rows;
			this.totalCount = totalCount;
		}

		public IEnumerator GetEnumerator() { return new
		  Core.TableEnumerator(rows); }

		public int TotalRowCount { get { return totalCount; } }

		public int RowCount { get { return rows.Count; } }

		public abstract int FieldCount { get; }

		public ArrayList Rows { get { return rows; } }

		private ArrayList rows;
		private int totalCount;
	}
}
```

Figure A4. AbstractDataTable.

APPENDIX B

SAMPLE CONCRETE DATA ACCESS CODE

Figure B1 displays the code for a stored procedure which retrieves a collection of *AdminPermission* records with paging, filtering and sorting functionality.

```
CREATE PROCEDURE dbo.a_AdminPermission_GetAdminRoleLinksPS
  @AdminRoleId int,
  @SortExp varchar(100),
  @PageSize int,
  @PageNum int,
  @SearchField varchar(25),
  @SearchKeyword varchar(25)
AS

CREATE TABLE #TempAdminPermission
(
        TempId int IDENTITY PRIMARY KEY,
        AdminPermission_Id int,
        selected bit
)

INSERT INTO #TempAdminPermission
(
        AdminPermission_Id,
        selected
)
EXEC
('
        SELECT
                AdminPermission.Id AS AdminPermission_Id,
                CAST (ISNULL(AdminRolePermissionLink.AdminRoleId, 0) as bit) AS
                  Selected
        FROM AdminPermission
        LEFT OUTER JOIN AdminRolePermissionLink ON
          AdminRolePermissionLink.AdminPermissionId = AdminPermission.Id AND
          AdminRolePermissionLink.AdminRoleId = ' + @AdminRoleId + '
        WHERE AdminPermission.' + @searchField + ' LIKE ''' + @searchKeyword + '%''
        ORDER BY ' + @SortExp
)

DECLARE @rows int
SET @rows = @@ROWCOUNT

DECLARE @first int
DECLARE @last int
SET @first = (@pageNum-1) * @pageSize + 1
SET @last = @first + @pageSize - 1
```

(figure continued)

```
SELECT
        AdminPermission_Id,
        selected
FROM #TempAdminPermission
WHERE TempId >= @first AND TempId <= @last

SELECT @rows
```

Figure B1. Sample Stored Procedure.

Figure B2 displays the code for a *DataClass* object which exposes methods operating on collections of records and methods, which do not require the instantiation of a single record class.

```
using System;
using System.Collections;
using System.Data;
using System.Data.SqlClient;

namespace Prssa.Data
{
        public class AdminPermissionCategoryData : Core.DataClass
        {
                #region constructor
                public AdminPermissionCategoryData() : base() {}
                #endregion

                #region GetList
                public AdminPermissionCategoryList GetList()
                {
                        SqlCommand command = new
                         SqlCommand("a_AdminPermissionCategory_GetList",
                           Connection);
                        command.CommandType = CommandType.StoredProcedure;
                        return LoadList(command);
                }
                #endregion

                #region GetRecords
                public AdminPermissionCategoryTable GetRecords(string SortExp)
                {
                        SqlCommand command = new
                           SqlCommand("a_AdminPermissionCategory_GetRecords",
                              Connection);
                        command.CommandType = CommandType.StoredProcedure;
                        command.Parameters.Add(new SqlParameter("@SortExp",
                           SortExp));
                        return LoadTable(command);
```

(figure continued)

```
}
#endregion

#region GetRecordsP
public AdminPermissionCategoryTable GetRecordsP(
        string SortExp,
        int PageSize,
        int PageNum)
{
        SqlCommand command = new
           SqlCommand("a_AdminPermissionCategory_GetRecordsP",
             Connection);
        command.CommandType = CommandType.StoredProcedure;
        command.Parameters.Add(new SqlParameter("@SortExp",
           SortExp));
        command.Parameters.Add(new SqlParameter("@PageSize",
           PageSize));
        command.Parameters.Add(new SqlParameter("@PageNum",
           PageNum));
        return LoadTable(command);
}
#endregion

#region GetRecordsPS
public AdminPermissionCategoryTable GetRecordsPS(
        string SortExp,
        int PageSize,
        int PageNum,
        string SearchField,
        string SearchKeyword)
{
        SqlCommand command = new
           SqlCommand("a_AdminPermissionCategory_GetRecordsPS",
             Connection);
        command.CommandType = CommandType.StoredProcedure;
        command.Parameters.Add(new SqlParameter("@SortExp",
           SortExp));
        command.Parameters.Add(new SqlParameter("@PageSize",
           PageSize));
        command.Parameters.Add(new SqlParameter("@PageNum",
           PageNum));
        command.Parameters.Add(new SqlParameter("@SearchField",
           SearchField));
        command.Parameters.Add(new SqlParameter("@SearchKeyword",
           SearchKeyword));
        return LoadTable(command);
}
#endregion

#region private

private AdminPermissionCategoryTable LoadTable(SqlCommand command)
{
        Connection.Open();
        SqlDataReader reader = command.ExecuteReader();
        ArrayList rows = new ArrayList();

        bool includeSelect = false;
        if (reader.FieldCount > 3)
                includeSelect = true;
```

(figure continued)

```
        while (reader.Read())
        {
                AdminPermissionCategoryRow r = new
                  AdminPermissionCategoryRow();
                rows.Add(r);

                if (reader[0] != System.DBNull.Value)
                        r.Id = reader.GetInt32(0);

                if (reader[1] != System.DBNull.Value)
                        r.Description = reader.GetString(1);

                if (reader[2] != System.DBNull.Value)
                        r.Rank = reader.GetInt32(2);

                r.Selected = false;
                if (includeSelect && reader.GetBoolean(3))
                        r.Selected = true;

        }
        int totalCount = rows.Count;
        if (reader.NextResult())
        {
                reader.Read();
                totalCount = reader.GetInt32(0);
        }
        reader.Close();
        Connection.Close();
        return new AdminPermissionCategoryTable(rows, totalCount);
}

private AdminPermissionCategoryList LoadList(SqlCommand command)
{
        Connection.Open();
        SqlDataReader reader = command.ExecuteReader();
        ArrayList rows = new ArrayList();

        bool includeSelect = false;
        if (reader.FieldCount > 2)
                includeSelect = true;

        while (reader.Read())
        {
                AdminPermissionCategoryListRow r = new
                  AdminPermissionCategoryListRow();
                r.Selected = false;
                rows.Add(r);

                if (reader[0] != System.DBNull.Value)
                        r.Id = reader.GetInt32(0);

                if (reader[1] != System.DBNull.Value)
                        r.Description = reader.GetString(1);

                r.Selected = false;
                if (includeSelect && reader.GetBoolean(2))
                        r.Selected = true;
```

(figure continued)

```
                        }
                        int totalCount = rows.Count;
                        if (reader.NextResult())
                        {
                                reader.Read();
                                totalCount = reader.GetInt32(0);
                        }
                        reader.Close();
                        Connection.Close();
                        return new AdminPermissionCategoryList(rows, totalCount);
                }

                #endregion
        }
}
```

Figure B2. Sample RecordData Object.

Figure B3 displays the code for a *DataClass* object which provides access to a single record, exposing its fields and the “update” method.

```
using System;
using System.Collections;
using System.Data;
using System.Data.SqlClient;
using Param = Prssa.Core.Parameter;

namespace Prssa.Data
{
        public class AdminPermissionCategory : Core.DataClass
        {
                #region constructor
                public AdminPermissionCategory(int recordId) : base()
                {
                        loadRecord(recordId);
                }
                #endregion

                #region int Id
                public int Id
                {
                        get { return id; }
                }
                #endregion
```

(figure continued)

```
                #region string Description
                public string Description
                {
                        get { return description; }
                }
                #endregion

                #region int Rank
                public int Rank
                {
                        get { return rank; }
                }
                #endregion

                #region private

                private void loadRecord(int recordId)
                {
                        SqlCommand command = new
                           SqlCommand("a_AdminPermissionCategory_GetRecord",
                             Connection);
                        command.CommandType = CommandType.StoredProcedure;
                        command.Parameters.Add(new SqlParameter("@Id", recordId));

                        Connection.Open();
                        SqlDataReader reader = command.ExecuteReader();
                        if (!reader.HasRows)
                                throw new Core.AppException("Record with id = " +
                                  recordId + " not found.");
                        reader.Read();

                        if (reader[0] != System.DBNull.Value)
                                id = reader.GetInt32(0);

                        if (reader[1] != System.DBNull.Value)
                                description = reader.GetString(1);

                        if (reader[2] != System.DBNull.Value)
                                rank = reader.GetInt32(2);

                        reader.Close();
                        Connection.Close();
                }
                private int id;
                private string description;
                private int rank;

                #endregion
        }
}
```

Figure B3. Sample Record Object.

Figure B4 displays the code for a *RecordField*, *RecordRow* and *RecordTable* object which are containers providing access to records and their fields.

```
using System;
using System.Collections;
using c = Prssa.Core;

namespace Prssa.Data
{
        public class AdminPermissionCategoryTable : c.AbstractDataTable
        {
                #region static IDataField IdField
                public static c.IDataField IdField { get { return new
                  IdFieldClass(); } }
                #endregion

                #region static IDataField DescriptionField
                public static c.IDataField DescriptionField { get { return new
                  DescriptionFieldClass(); } }
                #endregion

                #region static IDataField RankField
                public static c.IDataField RankField { get { return new
                  RankFieldClass(); } }
                #endregion

                #region static IDataField SelectedField
                public static c.IDataField SelectedField { get { return new
                  SelectedRecord(); } }
                #endregion

                #region constructor
                public AdminPermissionCategoryTable(ArrayList rows, int totalCount)
                  : base(rows, totalCount) {}
                #endregion

                #region int FieldCount
                public override int FieldCount { get { return 4; } }
                #endregion

                #region AdminPermissionCategoryRow this[int row]
                public AdminPermissionCategoryRow this[int row] { get { return
                  (AdminPermissionCategoryRow)Rows[row]; } }
                #endregion
```

(figure continued)

```
        #region private
        private class IdFieldClass : c.IDataField
        {
                public string DataField { get { return "Id"; } }
                public string SortExpression { get { return
                   "AdminPermissionCategory.Id"; } }
                public string Display { get { return "Id"; } }
        }

        private class DescriptionFieldClass : c.IDataField
        {
                public string DataField { get { return "Description"; } }
                public string SortExpression { get { return
                   "AdminPermissionCategory.Description"; } }
                public string Display { get { return "Description"; } }
        }

        private class RankFieldClass : c.IDataField
        {
                public string DataField { get { return "Rank"; } }
                public string SortExpression { get { return
                   "AdminPermissionCategory.Rank"; } }
                public string Display { get { return "Rank"; } }
        }

        private class SelectedRecord : c.IDataField
        {
                public string DataField { get { return "Selected"; } }
                public string SortExpression { get { return "Selected"; } }
                public string Display { get { return "Selected"; } }
        }
        #endregion
}

public class AdminPermissionCategoryRow : c.IDataRow
{
        #region int Id
        public int Id
        {
                get { return id; }
                set { id = value; }
        }
        #endregion

        #region string Description
        public string Description
        {
                get { return description; }
                set { description = value; }
        }
        #endregion

        #region int Rank
        public int Rank
        {
                get { return rank; }
                set { rank = value; }
        }
        #endregion
```

(figure continued)

```
                #region bool Selected
                public bool Selected
                {
                        get { return selected; }
                        set { selected = value; }
                }
                #endregion

                #region private
                private int id;
                private string description;
                private int rank;
                private bool selected;
                #endregion
        }
}
```

Figure B4. Sample Container Objects.

APPENDIX C

DATA DEFINITION LANGUAGE SCHEMA

```
<?xml version="1.0" encoding="utf-8" ?>
 <xs:schema id="test1" targetNamespace="codegenschema"
   elementFormDefault="qualified" xmlns="codegenschema"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="application">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="namespace" maxOccurs="unbounded">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="class" maxOccurs="unbounded">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="name" type="xs:string" />
           <xs:element name="type">
            <xs:simpleType>
             <xs:restriction base="xs:string">
              <xs:enumeration value="record" />
              <xs:enumeration value="link" />
              <xs:enumeration value="readonly" />
              <xs:enumeration value="final" />
             </xs:restriction>
            </xs:simpleType>
           </xs:element>
           <xs:element name="table">
            <xs:complexType>
             <xs:sequence>
              <xs:element name="name" type="xs:string" />
              <xs:element name="external" type="xs:boolean" minOccurs="0" />
              <xs:element name="field" maxOccurs="unbounded">
               <xs:complexType>
                <xs:all>
                 <xs:element name="name" type="xs:string" />
                 <xs:element name="sqldatatype" type="xs:string" />
                 <xs:element name="identity" type="xs:boolean" minOccurs="0" />
                 <xs:element name="primarykey" type="xs:boolean" minOccurs="0" />
                 <xs:element name="foreignkey" minOccurs="0" maxOccurs="1">
                  <xs:complexType>
                   <xs:sequence>
                    <xs:element name="reftable" type="xs:string" />
                    <xs:element name="reffield" type="xs:string" />
                   </xs:sequence>
                  </xs:complexType>
                 </xs:element>
                 <xs:element name="unique" type="xs:boolean" minOccurs="0" />
                 <xs:element name="encrypted" type="xs:boolean" minOccurs="0" />
                 <xs:element name="display" type="xs:string" minOccurs="0" />
                 <xs:element name="excludefromtable" type="xs:boolean"
                   minOccurs="0" />
```

(figure continued)

```
      <xs:element name="includewithparenttable" type="xs:boolean"
        minOccurs="0" />
      <xs:element name="includeinlist" type="xs:boolean" minOccurs="0"
        />
      <xs:element name="defaultsort" type="xs:boolean" minOccurs="0" />
      <xs:element name="readonlytype">
       <xs:simpleType>
        <xs:restriction base="xs:string">
         <xs:enumeration value="created" />
         <xs:enumeration value="modified" />
         <xs:enumeration value="timestamp" />
        </xs:restriction>
       </xs:simpleType>
      </xs:element>
     </xs:all>
    </xs:complexType>
   </xs:element>
   <xs:element name="additionalfield" minOccurs="0"
     maxOccurs="unbounded">
    <xs:complexType>
     <xs:all>
      <xs:element name="name" type="xs:string" />
      <xs:element name="sqldatatype" type="xs:string" />
      <xs:element name="sql" type="xs:string" />
      <xs:element name="sortexpression" type="xs:string" />
      <xs:element name="display" type="xs:string" minOccurs="0" />
      <xs:element name="excludefromtable" type="xs:boolean"
        minOccurs="0" />
      <xs:element name="includewithparenttable" type="xs:boolean"
        minOccurs="0" />
      <xs:element name="includeinlist" type="xs:boolean" minOccurs="0"
        />
      <xs:element name="defaultsort" type="xs:boolean" minOccurs="0" />
     </xs:all>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="additionalsproc" minOccurs="0" maxOccurs="unbounded">
 <xs:complexType>
  <xs:all>
   <xs:element name="name" type="xs:string" />
   <xs:element name="param" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="csdatatype" type="xs:string" />
      <xs:element name="encrypted" type="xs:boolean" minOccurs="0" />
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="returntype" type="xs:string" />
   <xs:element name="returnfield" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType mixed="true">
     <xs:all>
      <xs:element name="name" type="xs:string" minOccurs="0" />
      <xs:element name="csdatatype" type="xs:string" minOccurs="0" />
      <xs:element name="sortexpression" type="xs:string" minOccurs="0"
        />
```

(figure continued)

```
                <xs:element name="display" type="xs:string" minOccurs="0" />
               </xs:all>
              </xs:complexType>
             </xs:element>
            </xs:all>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" />
   <xs:attribute name="revised" type="xs:date" />
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure C1. Data Definition Language Schema.

APPENDIX D

SCHEMA FOR SAMPLE APPLICATION

```
<class>
 <name>Post</name>
 <type>record</type>
 <table>
  <field><name>Id</name></field>
  <field>
   <name>UserId</name>
   <foreignkey>
    <reftable>User</reftable>
    <reffield>Id</reffield>
   </foreignkey>
  </field>
  …other fields…
 </table>
</class>

<class>
 <name>User</name>
 <type>record</type>
 <table>
  <field><name>Id</name></field>
  …other fields…
 </table>
</class>

<class>
 <name>Permission</name>
 <type>readonly</type>
 <table>
  <field><name>Id</name></field>
  …other fields…
 </table>
</class>

<class>
 <name>UserPermissionLink</name>
 <type>link</type>
 <table>
  <field>
   <name>UserId</name>
   <foreignkey>
    <reftable>User</reftable>
    <reffield>Id</reffield>
   </foreignkey>
  </field>
  <field>
   <name>PermissionId</name>
   <foreignkey>
    <reftable>Permission</reftable>
    <reffield>Id</reffield>
   </foreignkey>
  </field>
 </table>
</class>
```

Figure D1. Schema for Sample Application.

APPENDIX E

LIST OF DATA ACCESS METHODS

Instance-related data object functionality

- ret record
- update record

Non-instance-related data object functionality

- create new record
- delete record
- get list
- get records
- get records with paging
- get records with paging and a filtering criteria

Non-instance-related data object functionality for each one-to-many relationship:

- get records by rel
- get records by rel with paging
- get records by rel with paging and a filtering criteria

Non-instance-related data object functionality for each many-to-many relationship:

- get records by link
- get records by link with paging
- get records by link with paging and a filtering criteria
- get links
- get links with paging
- get links with paging and a filtering criteria

Non-instance-related functionality for each many-to-many relationship:

- create link
- create all links by first data object
- create all links by second data object
- delete link
- delete links by first data object
- delete links by second data object

Figure E1. List of Data Access Methods.

APPENDIX F

WITNESS IDENTIFICATION APPLICATION SCHEMA

```
<?xml version="1.0" encoding="utf-8" ?>

<application
name="grad" revised="6/25/2006"
xmlns="codegenschema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="myschema file:schema.xsd">
<namespace>
<name>Data</name>

<class>
<name>LineupText</name>
<type>record</type>
<table>
<name>LineupText</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Name</name>
<sqldatatype>varchar(100)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Content</name>
<sqldatatype>varchar(4000)</sqldatatype>
</field>
<field>
<name>Rank</name>
<sqldatatype>int</sqldatatype>
</field>
<field>
<name>Modified</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>modified</readonlytype>
</field>
<field>
<name>ModifiedBy</name>
<sqldatatype>varchar(50)</sqldatatype>
</field>
</table>
</class>

<class>
<name>Suspect</name>
<type>record</type>
<table>
<name>Suspect</name>
<field>
```

(figure continued)

```
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>CaseId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Case</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>Gender</name>
<sqldatatype>char</sqldatatype>
</field>
<field>
<name>RaceId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Race</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>HairId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Hair</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>AgeId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Age</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>WeightId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Weight</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>Notes</name>
<sqldatatype>varchar(1000)</sqldatatype>
</field>
<field>
<name>Created</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>created</readonlytype>
</field>
<field>
<name>Modified</name>
<sqldatatype>datetime</sqldatatype>
```

(figure continued)

```
<readonlytype>modified</readonlytype>
</field>
<field>
<name>ModifiedBy</name>
<sqldatatype>varchar(50)</sqldatatype>
</field>
</table>
</class>

<class>
<name>Case</name>
<type>record</type>
<table>
<name>Case</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Number</name>
<sqldatatype>varchar(10)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
<includewithparenttable>true</includewithparenttable>
</field>
<field>
<name>Description</name>
<sqldatatype>varchar(1000)</sqldatatype>
</field>
<field>
<name>Created</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>created</readonlytype>
</field>
<field>
<name>Modified</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>modified</readonlytype>
</field>
<field>
<name>ModifiedBy</name>
<sqldatatype>varchar(50)</sqldatatype>
</field>
</table>
</class>

<class>
<name>Race</name>
<type>record</type>
<table>
<name>Race</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
```

(figure continued)

```
<name>Description</name>
<sqldatatype>varchar(25)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
<includewithparenttable>true</includewithparenttable>
</field>
</table>
</class>

<class>
<name>Hair</name>
<type>record</type>
<table>
<name>Hair</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Description</name>
<sqldatatype>varchar(25)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
<includewithparenttable>true</includewithparenttable>
<display>Color</display>
</field>
</table>
</class>

<class>
<name>Age</name>
<type>record</type>
<table>
<name>Age</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Description</name>
<sqldatatype>varchar(25)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
<includewithparenttable>true</includewithparenttable>
<display>Range</display>
</field>
</table>
</class>
<class>
<name>Weight</name>
<type>record</type>
<table>
<name>Weight</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
```

(figure continued)

```
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Description</name>
<sqldatatype>varchar(25)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
<includewithparenttable>true</includewithparenttable>
<display>Range</display>
</field>
</table>
</class>

<class>
<name>Lineup</name>
<type>record</type>
<table>
<name>Lineup</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>SuspectId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Suspect</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>SuspectPhotoPosition</name>
<sqldatatype>int</sqldatatype>
</field>
<field>
<name>CaseId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Case</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>Description</name>
<sqldatatype>varchar(500)</sqldatatype>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Created</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>created</readonlytype>
</field>
<field>
<name>Modified</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>modified</readonlytype>
</field>
```

(figure continued)

```
<field>
<name>ModifiedBy</name>
<sqldatatype>varchar(50)</sqldatatype>
</field>
</table>
</class>

<class>
<name>Photo</name>
<type>record</type>
<additionalsproc>
<name>GetByCriteria</name>
<returntype>generate</returntype>
<param>
<name>Query</name>
<csdatatype>string</csdatatype>
</param>
<returnfield>Id</returnfield>
</additionalsproc>
<table>
<name>Photo</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>ExternalId</name>
<sqldatatype>varchar(20)</sqldatatype>
<unique>true</unique>
</field>
<field>
<name>Gender</name>
<sqldatatype>char</sqldatatype>
</field>
<field>
<name>RaceId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Race</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>HairId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Hair</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>AgeId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Age</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
```

(figure continued)

```
<name>WeightId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Weight</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>Created</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>created</readonlytype>
</field>
<field>
<name>Modified</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>modified</readonlytype>
</field>
<field>
<name>ModifiedBy</name>
<sqldatatype>varchar(50)</sqldatatype>
</field>
</table>
</class>

<class>
<name>LineupPhotoLink</name>
<type>link</type>
<table>
<name>LineupPhotoLink</name>
<field>
<name>LineupId</name>
<sqldatatype>int</sqldatatype>
<primarykey>true</primarykey>
<foreignkey>
<reftable>Lineup</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>PhotoId</name>
<sqldatatype>int</sqldatatype>
<primarykey>true</primarykey>
<foreignkey>
<reftable>Photo</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
</table>
</class>

<class>
<name>LineupView</name>
<type>record</type>
<additionalsproc>
<name>Finalize</name>
<returntype>int</returntype>
<param>
<name>LineupViewId</name>
<csdatatype>int</csdatatype>
</param>
<param>
```

(figure continued)

```
<name>Relevance</name>
<csdatatype>string</csdatatype>
</param>
</additionalsproc>
<table>
<name>LineupView</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>LineupId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Lineup</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>WitnessFirstName</name>
<sqldatatype>varchar(25)</sqldatatype>
</field>
<field>
<name>WitnessLastName</name>
<sqldatatype>varchar(25)</sqldatatype>
</field>
<field>
<name>Relevance</name>
<sqldatatype>varchar(4000)</sqldatatype>
<excludefromtable>false</excludefromtable>
</field>
<field>
<name>Created</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>created</readonlytype>
</field>
<field>
<name>CreatedBy</name>
<sqldatatype>varchar(50)</sqldatatype>
</field>
<field>
<name>IsCompleted</name>
<sqldatatype>bit</sqldatatype>
</field>
<additionalfield>
<name>FullName</name>
<sqldatatype>varchar(50)</sqldatatype>
<sql>[LineupView].WitnessLastName + ', ' + [LineupView].WitnessFirstName AS
LineupView_FullName</sql>
<display>Witness Name</display>
<sortexpression>WitnessLastName</sortexpression>
<includewithparenttable>true</includewithparenttable>
<defaultsort>true</defaultsort>
</additionalfield>
</table>
</class>
<class>
<name>PhotoView</name>
<type>final</type>
```

(figure continued)

```
<table>
<name>PhotoView</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
</field>
<field>
<name>LineupViewId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>LineupView</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>PhotoId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Photo</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>SuspectId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>Suspect</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>IsSuspect</name>
<sqldatatype>bit</sqldatatype>
</field>
<field>
<name>Result</name>
<sqldatatype>varchar(10)</sqldatatype>
</field>
<field>
<name>Certainty</name>
<sqldatatype>varchar(1000)</sqldatatype>
</field>
</table>
</class>

<class>
<name>User</name>
<type>record</type>
<additionalsproc>
<name>ValidateUser</name>
<returntype>int</returntype>
<param>
<name>Login</name>
<csdatatype>string</csdatatype>
</param>
<param>
<name>Password</name>
<csdatatype>string</csdatatype>
<encrypted>true</encrypted>
</param>
```

(figure continued)

```
</additionalsproc>
<additionalsproc>
<name>LogUser</name>
<returntype>void</returntype>
<param>
<name>UserId</name>
<csdatatype>int</csdatatype>
</param>
</additionalsproc>
<table>
<name>User</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Login</name>
<sqldatatype>varchar(50)</sqldatatype>
<unique>true</unique>
</field>
<field>
<name>Password</name>
<sqldatatype>varbinary(16)</sqldatatype>
<excludefromtable>false</excludefromtable>
<encrypted>true</encrypted>
</field>
<field>
<name>FirstName</name>
<sqldatatype>varchar(25)</sqldatatype>
</field>
<field>
<name>LastName</name>
<sqldatatype>varchar(25)</sqldatatype>
</field>
<field>
<name>Created</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>created</readonlytype>
</field>
<field>
<name>Modified</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>modified</readonlytype>
</field>
<field>
<name>ModifiedBy</name>
<sqldatatype>varchar(50)</sqldatatype>
</field>
<additionalfield>
<name>FullName</name>
<sqldatatype>varchar(50)</sqldatatype>
<sql>[User].LastName + ', ' + [User].FirstName AS User_FullName</sql>
<display>Full Name</display>
<sortexpression>LastName</sortexpression>
<includeinlist>true</includeinlist>
<includewithparenttable>true</includewithparenttable>
<defaultsort>true</defaultsort>
</additionalfield>
</table>
</class>
<class>
```

(figure continued)

```
<name>Role</name>
<type>record</type>
<table>
<name>Role</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Description</name>
<sqldatatype>varchar(25)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
</field>
</table>
</class>

<class>
<name>Permission</name>
<type>readonly</type>
<additionalsproc>
<name>GetPermissionCodesByUser</name>
<returntype>ArrayList</returntype>
<param>
<name>UserId</name>
<csdatatype>int</csdatatype>
</param>
</additionalsproc>
<additionalsproc>
<name>GetAllRecordsByPermCatByRole</name>
<param>
<name>CategoryId</name>
<csdatatype>int</csdatatype>
</param>
<param>
<name>RoleId</name>
<csdatatype>int</csdatatype>
</param>
<returntype>generate</returntype>
<returnfield>Id</returnfield>
<returnfield>Description</returnfield>
<returnfield>
<name>Selected</name>
<csdatatype>bool</csdatatype>
<sortexpression>Selected</sortexpression>
<display>Selected</display>
</returnfield>
</additionalsproc>
<table>
<name>Permission</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>CategoryId</name>
```

(figure continued)

```
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>PermissionCategory</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>Description</name>
<sqldatatype>varchar(100)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
<defaultsort>true</defaultsort>
</field>
<field>
<name>Rank</name>
<sqldatatype>smallint</sqldatatype>
</field>
</table>
</class>

<class>
<name>PermissionCategory</name>
<type>readonly</type>
<table>
<name>PermissionCategory</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
<defaultsort>true</defaultsort>
</field>
<field>
<name>Description</name>
<sqldatatype>varchar(50)</sqldatatype>
<unique>true</unique>
<includeinlist>true</includeinlist>
</field>
<field>
<name>Rank</name>
<sqldatatype>smallint</sqldatatype>
</field>
</table>
</class>

<class>
<name>UserRoleLink</name>
<type>link</type>
<table>
<name>UserRoleLink</name>
<field>
<name>UserId</name>
<sqldatatype>int</sqldatatype>
<primarykey>true</primarykey>
<foreignkey>
<reftable>User</reftable>
<reffield>Id</reffield>
</foreignkey>
```

(figure continued)

```
</field>
<field>
<name>RoleId</name>
<sqldatatype>int</sqldatatype>
<primarykey>true</primarykey>
<foreignkey>
<reftable>Role</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
</table>
</class>
<class>
<name>RolePermissionLink</name>
<type>link</type>
<table>
<name>RolePermissionLink</name>
<field>
<name>RoleId</name>
<sqldatatype>int</sqldatatype>
<primarykey>true</primarykey>
<foreignkey>
<reftable>Role</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>PermissionId</name>
<sqldatatype>int</sqldatatype>
<primarykey>true</primarykey>
<foreignkey>
<reftable>Permission</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
</table>
</class>

<class>
<name>UserCaseLink</name>
<type>link</type>
<table>
<name>UserCaseLink</name>
<field>
<name>UserId</name>
<sqldatatype>int</sqldatatype>
<primarykey>true</primarykey>
<foreignkey>
<reftable>User</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>CaseId</name>
<sqldatatype>int</sqldatatype>
<primarykey>true</primarykey>
<foreignkey>
<reftable>Case</reftable>
<reffield>Id</reffield>
</foreignkey>
```

(figure continued)

```
</field>
</table>
</class>
<class>
<name>UserLog</name>
<type>final</type>
<table>
<name>UserLog</name>
<field>
<name>Id</name>
<sqldatatype>int</sqldatatype>
<identity>true</identity>
<primarykey>true</primarykey>
<includeinlist>true</includeinlist>
</field>
<field>
<name>UserId</name>
<sqldatatype>int</sqldatatype>
<foreignkey>
<reftable>User</reftable>
<reffield>Id</reffield>
</foreignkey>
</field>
<field>
<name>Created</name>
<sqldatatype>datetime</sqldatatype>
<readonlytype>created</readonlytype>
</field>
</table>
</class>

</namespace>
</application>
```

Figure F1. Witness Identification Application Schema.