

**WASTE TIRE TRACKING WITH THE  
ENVIRONMENTAL DATA EXCHANGE**

A thesis written at

**MISSISSIPPI DEPARTMENT OF ENVIRONMENTAL QUALITY**

and submitted to

**KETTERING UNIVERSITY**

in partial fulfillment  
of the requirements for the  
degrees of

**BACHELOR OF SCIENCE IN COMPUTER SCIENCE**

**and**

**BACHELOR OF SCIENCE IN COMPUTER ENGINEERING**

by

**CHRISTOPHER J. LIEB**

December 2008

---

Author

---

Employer Advisor

---

Faculty Advisor

## **DISCLAIMER**

This thesis is submitted as partial and final fulfillment of the cooperative work experience requirements of Kettering University needed to obtain a Bachelor of Science in Computer Science and Computer Engineering Degree.

The conclusions and opinions expressed in this thesis are those of the writer and do not necessarily represent the position of Kettering University or the Mississippi Department of Environmental Quality, or any of its directors, officers, agents, or employees with respect to the matters discussed.

## PREFACE

This thesis represents the capstone of my four and a half years combined academic work at Kettering University and job experience at UPS and the Mississippi Department of Environmental Quality. Academic experiences in Computer Science and Computer Engineering proved to be valuable assets while I developed this thesis and addressed the problem it concerns.

Although this thesis represents the compilation of my own efforts, I would like to acknowledge and extend my sincere gratitude to the following persons for their valuable time and assistance, without whom the completion of this thesis would not have been possible:

- 1 Mr. Christopher Staten, Senior Business Systems Analyst at the Mississippi Department of Environmental Quality, for extending any and all help which I have needed throughout my time at the Mississippi Department of Environmental Quality, as well as throughout this entire thesis process.
- 2 Dr. James Huggins, Associate Professor of Computer Science at Kettering University, for his direction and guidance throughout this thesis project.
- 3 My parents, Sam and Peggy Lieb, for the love and support they have given me throughout my academic experiences at Kettering University, and the entire time that I have worked on this thesis.

## TABLE OF CONTENTS

DISCLAIMER . . . . .	ii
PREFACE . . . . .	iii
I. INTRODUCTION . . . . .	1
Problem Topic . . . . .	1
Background . . . . .	2
Environmental Data Exchange . . . . .	2
Waste tire . . . . .	2
Criteria and Parameter Restrictions . . . . .	3
Methodology . . . . .	4
Assessment phase . . . . .	4
Refactoring phase . . . . .	5
Planning phase . . . . .	5
Implementation phase . . . . .	5
Evaluation phase . . . . .	5
Primary Purpose . . . . .	6
Overview . . . . .	6
II. THE DATABASE . . . . .	7
Referential Integrity . . . . .	7
Database Schema Cruft . . . . .	10
Stored Procedures . . . . .	11
III. THE APPLICATION . . . . .	14
Mixed View and Controller . . . . .	14
Inconsistent Appearance . . . . .	16
Logging . . . . .	17
IV. CONCLUSIONS AND RECOMMENDATIONS . . . . .	21
Conclusions . . . . .	21
Issues Encountered . . . . .	21
Recommendations . . . . .	22
APPENDICES . . . . .	23
APPENDIX A: ABET PROGRAM OUTCOMES - COMPUTER SCIENCE . . . . .	24

APPENDIX B: ABET PROGRAM OUTCOMES - COMPUTER ENGINEER-  
ING . . . . . 26

## I. INTRODUCTION

The primary goal of the Mississippi Department of Environmental Quality (MDEQ) is to protect the environment of the state of Mississippi. To help do this, MDEQ has provided ways for people of the state to communicate with the agency, one of the newest being through tools on the Internet. The Environmental Data Exchange (enDx) is one of these tools. However, not all communication is currently possible using the current set of MDEQ websites, leaving paper as the only form of communication. One case of this is communications regarding waste tire permitting and monitoring.

### **Problem Topic**

Currently, enDx is under-utilized. Most of the job that enDx handles is still being performed predominantly via paper, even though an electronic version is available. This is not satisfactory since it is much quicker to submit and process reports via enDx.

At the moment, waste tire permitting and reporting is handled exclusively with paper. Due to the large number of people participating and the large number of reports filed, the small group tasked with regulating waste tire handling is unable to process all of the information that they receive. This prevents them from properly regulating the industry.

## **Background**

### **Environmental Data Exchange**

enDx is a J2EE-based web application that was written by a contracting company in Atlanta. About four years ago, MDEQ let the contract with the contracting company expire. This allowed enDx to be maintained in-house with lower maintenance costs. Chris Staten was the only developer that met with members of the contracting firm when the code was transferred to MDEQ. This has made him the sole authority for problems with enDx, since he is the only one that understands any part of its inner workings.

It turns out that the codebase for enDx is poorly structured and lacks any documentation. A quick examination of the database schema reveals that there are no foreign key constraints and few tables with primary keys within the database, leaving no clue regarding the exact relations between the different entities in the database. MDEQ staff are only able to infer relationships by matching the names of fields that appear to be IDs in different tables.

enDx is currently used only for construction notices of intent (CNOIs) and water annual inspection reports. Due to the low visibility of the site and its narrow focus, enDx receives little traffic. Management has been trying to figure out how to drive more traffic to enDx to lessen the load of paper documents that must be processed by the agency.

### **Waste tire**

The waste tire industry in Mississippi is a large business. It involves companies that “produce” waste tires, companies that transport (haul) waste tires, and companies that process the waste tires, either by recycling them, retreading them, or storing them for later recycling. It is part of MDEQ’s job to monitor the waste tire industry’s participants to make sure that they are not polluting and that MDEQ knows of their activity.

One of the issues that faces MDEQ is waste tire dumping. This happens when a waste tire hauler dumps the tires he is hauling somewhere other than at a processing

facility. By doing this, the hauler can earn more money, since he does not have to pay any fees at the processing facility, or pay for the fuel to travel to the processing facility.

Waste tire monitoring is handled by the waste tire group of the Environmental Permitting Division (EPD). The primary responsibility of the group is to process permit applications submitted by waste tire haulers and processors. Both types of entities must apply for permits annually. This job, by itself, is not too time consuming.

The real work for the waste tire group is processing the reports submitted by the haulers and processors. The haulers must submit annual reports detailing the number of tires that they hauled and to which facilities they were hauled. The processors must submit monthly reports detailing how many tires they received and which haulers delivered them.

Due to the large amount of paperwork that the waste tire group must process and the small number of staff allotted to the group, it is impossible to process all of the paperwork. This means that the group is unable to detect inconsistencies in reported numbers and act on those inconsistencies. Because of this, it is very difficult, if not impossible, to detect waste tire dumpers.

### **Criteria and Parameter Restrictions**

The first step to improving enDx is documenting it so that MDEQ staff know how its internals work. Once this is done, it will be much easier to refactor the code to meet MDEQ standards. Source-level documentation for enDx is desired, though MDEQ would like more in-depth documentation. While in-depth documentation will exist for new code, it will be more difficult to create documentation for the existing code.

The second step to improving enDx will be general code clean-up. This will include using an actual logging facility (Apache Commons Logging) instead of the mishmash of methods currently used, converting the code to make use of new language features added in Java 5, rewriting code that makes use of deprecated methods, writing unit tests for existing code, making all of the web pages HTML 4.01 Strict compliant, and

refactoring business logic out of the web pages and placing it into servlets and tag libraries.

Once this is completed, the current process used by the waste tire group will be integrated into enDx. Existing functionality will be used to handle permit applications and approvals. A new portion of the application will handle the display of reports built from submitted data.

This project will not involve the porting of the application to another language or operating system, so it will remain in Java. The application will be updated to Java 2 Enterprise Edition (J2EE) 1.4 and Java 2 Standard Edition (J2SE) 5 from J2EE 1.3 and J2SE 1.3; its target environment will be updated from IBM WebSphere Application Server 5.1 to version 6.1.

## **Methodology**

To meet the aforementioned criteria, the process has been divided into the following five phases: assessment, refactoring, planning, implementation, and evaluation. This thesis details the first three phases of the project. The fourth and fifth phases are not covered due to time and staffing constraints.

### **Assessment phase**

The project will begin with an assessment of the current state of the application. This will include working out the exact functionality that is currently available. This will form the basis for documentation of the program as well as providing a checklist of functionality that should still work at the completion of the project.

Basic models of the system will be created as a basis for future work. These will include static class models of the program and entity-relation diagrams of the database.

### **Refactoring phase**

This phase will include basic code cleanup as well as documentation of all classes and methods. The database schema will be updated to add primary keys and foreign key relationships to help with consistency in the database.

### **Planning phase**

Once documentation for the existing code has been created, planning for the changes to add the requested new functionality to the application will begin. This will include changes to the database schema, the static class model, and the web pages. The plans will include complete static class models, as well as functional class models and detailed use cases.

### **Implementation phase**

The first part of this phase will be to update the database schema and to write SQL scripts to migrate the existing data to the new schema, if such a migration is not trivial.

A new API will be created over the database using Hibernate so that raw SQL can be removed from the application code. This should help prevent any possible SQL injection attacks on the web application, as well as making database queries more readable and less likely to cause database errors or data corruption.

Once the API is created, existing code will be ported to use the new API. Other work can be done while this is happening, since code can be replaced piecemeal without breaking anything.

As the application is being transitioned to the new database API, the new functionality for waste tire reporting will be implemented.

### **Evaluation phase**

After the programming is done, a check list of requirements will be reviewed to ensure that the application meets all of the original requirements. Among these

requirements will be that all previous functionality from enDx still work as it originally did (unless a design decision was made to alter specific behavior), that all of the new functionality for waste tire permitting and reporting work correctly, and that the database migration scripts don't corrupt or destroy data unintentionally.

### **Primary Purpose**

The purpose of this project is to clean up and document enDx and then add functionality to it to support waste tire permitting and reporting.

### **Overview**

This thesis details the analysis of enDx, the gathering of requirements for waste tire, the refactoring and documentation of enDx, and the path that MDEQ should follow to complete the additions to enDx and evaluate its effectiveness. By the end of this thesis, the groundwork should be laid for the improvement of enDx, with a clear path toward implementation.

## II. THE DATABASE

### Referential Integrity

One aspect of good database design is the normalization of the data model to prevent unnecessary duplication of data. As part of normalization, data needs to be linked in some way to establish a relation amongst the data. This is normally accomplished by adding a unique identifier to a record in one table (a primary key), then having a record in another table reference that identifier (a foreign key). Relational database engines can help keep the relationships between primary keys and foreign keys consistent by making sure that primary keys are unique and that foreign keys do not reference undefined primary keys.

In enDx, fields that hold the data for primary and foreign keys exist, but the database engine does not know of their existence, and therefore cannot help keep data consistent. This creates many problems, including duplicate primary keys and foreign keys that do not reference any information.

In a table, all primary keys are required to be unique for referential integrity to be established. If this requirement is not met, then a reference to a duplicate primary key cannot be resolved to a single record, preventing a link from being established. Some of the tables in the enDx schema had duplicated identifiers in their primary key fields, which could have caused issues. This situation also prevented the database engine from being able to mark the column as a primary key, which means that the database engine could not help in establishing indexes or ensuring the uniqueness of identifiers in the column.

In a table, all foreign keys are required to refer to a valid primary key for integrity

to be established. If this requirement is not met, then foreign keys cannot be guaranteed to point to existent data. In the enDx schema, no foreign keys were established, instead putting all of the logic to maintain them directly in the application. Due to mistakes in the implementation of the application and people editing data in the database by hand instead of using the application, some foreign keys in the database referenced data that had been deleted at some time in the past. Not only that, but there is little information on what field each foreign key was referencing, making it even harder to check the integrity of the references.

To fix these problems, all relationships within the database had to be determined. Once this was done, data types of columns that would be linked were changed, since foreign keys must be of the same data type as the primary keys they reference. All of the potential primary keys were inspected for duplicates. Duplicate primary keys were discovered in three tables. These duplicates were caused by data being inserted into the table without using the sequence defined in the database to generate the new identifier. Because the sequence was not used all of the time in these tables, the sequence was generating identifiers that had already been used. While the code that was improperly inserting data into these tables was not found, the duplicated identifiers were resolved. Since none of the tables which had duplicate identifiers were ever referenced by other tables, the sequence generators for those tables were reset, and then all of the identifiers were reinitialized, assuring that all identifiers were unique and that the sequence generators were in the correct state.

After fixing the primary key fields, we had to fix the foreign key fields. Many of the foreign key fields contained illegal data, such as keys that no longer existed or empty strings instead of NULLs. All illegal values were changed to NULLs. References to non-existent data were resolved one in one of two ways: recreating the data that was being referenced, or deleting the data that was referencing the non-existent data. In a few cases, the missing data was reconstructed by looking at the code in the user interface. This was

possible only in tables that acted as lookup data for the UI, which led to values being hard coded in the application rather than pulled from the database. The remaining data was determined to be expendable since it dated from when the application was first deployed, and was likely test data that was never deleted.

One table, DEQ\_ADDRESS, provided a unique challenge. It contained a column, ADDR\_ID, that at first appeared to be the primary key for the table. The data in the column, however, appeared to be in varying formats. This led to the conclusion that the data stored in ADDR\_ID was not primary key information, but, rather, data pulled in from other tables. Four tables were capable of being joined against some of the data in ADDR\_ID: DEQ\_DOCUMENT, DEQ\_USER, DEQ\_PERSON, and DEQ\_FACILITY. Creating foreign keys on ADDR\_ID for each of the four relationships yielded two problems: first, DOCUMENT\_ID, the primary key of DEQ\_DOCUMENT, was of a different data type than the primary keys of the other three tables and ADDR\_ID; second, for each foreign key there would be a large number of values that would not match, preventing it from being created. To get around this, four new columns in DEQ\_ADDRESS were created, one for each of the foreign keys. A newly-created trigger will automatically copy a new value from ADDR\_ID into the correct new field to maintain referential integrity across those five tables.

After the data and schema were fixed, primary keys and foreign keys were created within the database. Database triggers were also created to make sure that none of the primary keys could get out of sync with their sequence generators again.

Once changes to the database schema were complete, the current version of enDx was tested against the new schema to make sure that no errors were introduced while making our changes. Tests were not performed against the development version of enDx since it is not scheduled to reach production status in the near future.

Testing quickly revealed a deadlocking issue when trying to view certain reports within enDx. With the help of the database administrator, the problem was isolated to two

stored procedures: one that was modifying data in a table and one that was reading data out of a different table. The table that was being modified shared a reference with the table that was being read. These two stored procedures were somehow managing to deadlock each other due to them both accessing the same primary key.

The cause of the deadlocks was a stored procedure that was modifying data. Instead of committing its changes right after it made them, it was relying on the auto-commit feature of the database engine. Because of this, the database engine was getting to decide when to commit the data. This meant that a new stored procedure was being run before the data was committed to the database. Since the data had not been committed yet, a lock was still held to the table being modified, preventing the other stored procedure from being able to access the table.

To fix this issue, all of the stored procedures were changed to have an explicit commit of changes before they exit, rather than relying on auto-commit. Extensive testing has not revealed any more deadlock issues.

### **Database Schema Cruft**

While designing a system, chances are that the design will change as the system grows and is developed. This can cause a buildup of old code that is no longer used but still resides within the code base. Evidence of this cruft building up within enDx are most evident in the enDx schema.

Many tables in the schema have columns that are completely empty, and there are even a few tables that are completely empty. While empty tables do not affect the performance of the application, they do require extra effort for documentation and maintenance. The more important issue is the occurrence of unused columns within some of the tables.

While an empty table takes up little more storage than just the table's metadata, a populated table with unused columns has the possibility of consuming much more space

than it would require without the columns in the table. While the database engine might be able to optimize the storage of a row with empty fields, it might sacrifice speed to do so since each row would not be the same length. A similar effect can be observed by using variable length character (VARCHAR) fields, which are not a fixed length, in a table. When they are used, speed is lost when reading and writing data, but storage space is saved since only enough space is used to store the data.

One instance of an unused table is the DEQ\_COMP\_AIR table, which is used to hold data on air permits. While the original requirements for enDx specified that it should be able to handle air permits, this functionality was never implemented within enDx. However, a set of tables were still created within the schema to hold the data for the air permitting functionality. Because of this, there are four tables within the enDx schema, including DEQ\_COMP\_AIR, that are never used. Since the air permitting functionality may be implemented in the future, the tables in the schema were allowed to remain and were given the same treatment as the rest of the tables when it came to adding primary and foreign keys.

Unused columns were rather common in the enDx schema. Since these had the possibility of causing larger performance issues than the unused tables, extra analysis was performed. While no information was being inserted by the enDx application into these columns, they weren't referenced in any stored procedure that touched the table they were in, and were also coded into the application. While the unused columns were not removed at this time, they will be considered for removal at a later date.

### **Stored Procedures**

One choice that was made when enDx was being developed was to put the data logic into the database instead of a separate layer that would sit between the application and the database. In this case, the logic was placed into stored procedures. While Oracle, the database engine that enDx runs on, allows the creation of stored procedures in

PL/SQL, its own variant of SQL, it also allows stored procedures to be written in Java. The stored procedures in enDx were authored in Java, with some PL/SQL wrapper code so that they could be loaded into Oracle.

Upon inspection, it was discovered that the stored procedures were all coded differently, even though they only performed one of four functions: retrieve, insert, update, or delete data. The vast majority of the stored procedures only touched one or two tables. This means that the stored procedures could share a similar structure, making them easier to maintain since there would be a single way that they would work. One of the largest issues they had were the methods that they used to handle errors.

Oracle allows stored procedures to be written in Java using the JDBC (Java database connectivity) library, which is part of any Java Standard Edition distribution. The only difference between writing a normal function in Java using JDBC and an Oracle Java stored procedure using JDBC is that Oracle provides a static, globally scoped connection to the database for the programmer to use. This means that one still has to handle connection exceptions that are exposed through JDBC, as well as a host of other JDBC and Java exceptions.

The original developer of the application decided to swallow exceptions that were thrown by the stored procedures. This is not a good practice, since there is no way to tell whether a stored procedure was able to execute successfully. Unfortunately, not all exceptions were swallowed, or were handled in odd ways, preventing anything meaningful from being done in the case of an exception occurring in a stored procedure. Not only that, but since exceptions did not leave the database, it was very difficult to fix errors coming from the stored procedures since all that was reported was a generic database error that contained no information.

To fix this situation, standard templates were created for each type of stored procedure. These templates handled exceptions by allowing them to bubble up to the database engine, since they cannot be handled from within the stored procedure. Also, by

allowing the exceptions to bubble up to the database engine, it became much easier to diagnose problems in the stored procedures, since the errors were logged in the database logs as well as being reported to the application as a database error.

Once the templates were created, each stored procedure was modified to follow the template that applied to it as closely as possible. This was a fairly straight-forward procedure since all of the stored procedures could be easily fit to the templates.

### III. THE APPLICATION

#### Mixed View and Controller

When Java first moved to the World Wide Web, it did so through the technology of servlets. While the better known applets are bits of Java that run on the client-side in a browser, servlets are bits of Java that run server-side inside a servlet container, which is an application server for hosting Java web applications. Servlets are nothing more than Java classes that extend the servlet class and inherit from the `HttpServlet` interface. They define methods that are executed on the different HTTP requests (GET, POST, etc.) and are mapped to world-accessible URLs using an XML deployment descriptor. In order to create a web page using servlets, code must write strings of HTML into an output writer. After the invoked method has completed execution, the contents of the writer are flushed to the client that requested the page. The biggest problem with this model is that, since the HTML is contained in strings, it is not easy to validate the HTML in the servlet without invoking the servlet with every possible set of data and testing the resulting HTML output.

To fix this, a templating language, called Java Server Pages (JSP), was added to Java 2 Enterprise Edition 2.1. JSP is an XML-based templating language, which makes it much easier to validate the HTML that might be produced before the JSP is invoked. JSPs work by being compiled into a servlet and then into Java bytecode dynamically when they are first executed by the servlet engine. This is in contrast to a regular servlet, which would be compiled into Java bytecode when the rest of the web application is being compiled, before the web application is deployed to a servlet container. The servlet that is generated from the JSP will look much the same as the servlet that would have been coded

by hand, but with the added advantage that it is possible to catch many errors in the markup before assembling and deploying the application.

The first specification of JSP contained the scriptlet tag, which allowed Java code to be embedded directly into a JSP. This prevented programmers from having to write servlets, since they could put data retrieval and processing logic directly into JSPs. While this might make things easier for beginners, it has many drawbacks.

The first drawback is that the Java code in the JSP is not compiled until the JSP is invoked for the first time by the servlet container. This means that you could easily deploy a JSP with syntactically incorrect Java code and not know it until someone hits the JSP and gets a stack trace or error page instead of the desired web page. If the code had been placed into a servlet instead of a JSP, the incorrect code could have been caught by the compiler or by unit tests long before the servlet was deployed to an application server.

The second drawback is that, by placing data processing code into the display template, the model-view-controller (MVC) model is violated. In this model, the servlet is the controller and the JSP is the view. The view's responsibility is to get input from the user or to display data to the user; the controller's responsibility is to process data received from the view or to retrieve data for the view to display. When the Java code that processes data is placed directly into a JSP, it can cause confusion since one can easily intermingle data processing code with display logic. This makes maintenance much harder since the code is trying to do multiple, unrelated functions at once.

At first, JSPs were almost useless without scriptlets since JSP had no built-in facility for conditional or iterative execution. To help alleviate this issue, tag libraries, specifically the Java Standard Tag Library (JSTL), were added to the JSP 1.2 specification. Tag libraries allow custom XML tags to be added to a JSP, allowing for conditional and iterative logic, along with whatever else an author may wish to add. With the addition of JSTL, the need for scriptlets in a proper MVC-based JSP were removed.

Since enDx was written right around the time that JSTL was released, it relied on

scriptlets to implement much of the display logic in its JSPs. The application also intermingles display logic and controller functionality in places, which violates the MVC model. To make the JSPs easier to maintain in the future, it was decided to move any controller logic in the JSPs into their controlling servlets and to convert the remaining display logic to use JSTL and custom tags so that the scriptlets can be completely removed from the JSPs. This will result in the JSPs being pure XML, without any Java code or scriptlets remaining.

### **Inconsistent Appearance**

When enDx was written, most styling of the web interface was done through HTML style attributes instead of CSS. This led to large amounts of duplicate code for layout and styling of the content. This duplication made it very easy to accidentally change the appearance of one page without realizing it. This has resulted in each page having a slightly different appearance as these hard-coded HTML styles were accidentally altered.

To fix this, the page header and footer that were common amongst the pages were extracted and placed into custom JSP tags. Doing this allowed all of the header and footer code to be replaced by two custom tags. All presentational markup, including layout tables, was replaced with syntactically-correct HTML elements and styled them with CSS. These two custom tags were added to a larger library of custom tags to form a template for enDx.

The custom tags created for the template were page, head, body, header, and footer. Page was responsible for outputting the doctype (HTML 4.01 Strict) and the `<html>` tags. Head was responsible for outputting the contents of the `<head>` tag, including being able to include extra JavaScript and CSS files if they were needed. Body was responsible for outputting the `<body>` tags. Since the only children of `<html>` are `<head>` and `<body>`, custom tags were created to replace both so that the first two levels

of the HTML document would be overridden consistently. Header was responsible for outputting the page header, including a logout link whose display can be toggled with a single attribute of the header custom tag. Footer was responsible for outputting the page footer. Footer was the only tag that was non-deterministic since it would output the current year for the copyright statement without requiring the user of the tag to specify the date.

Once the template library was created, each JSP was converted to use the new library. This brought an immediate improvement to the appearance of the pages since they now had identical headers and footers. There were a few static HTML pages that were converted to JSPs so that the new templates could be used with them.

Since doctypes have been added to all of the pages that are served, validation errors can be easily fixed. One type of validation error that will be encountered by using a strict doctype is the use of HTML styles. Strict doctypes deprecate or remove most of the HTML presentational attributes and elements, forcing one to use CSS for styling page elements. By using the strict doctype, all pages will also trigger the standards compliance or almost-standard compliance mode in all common web browsers. This means that the pages should now be laid out consistently across browsers when using CSS styles. Without the strict doctype, most browsers will switch to quirks mode, rendering the page similar to their predecessors, which were not consistent with one another. Having the pages render under standards mode should make a transition to a CSS-styled site much easier since the differences in rendering between browsers will be minimized.

## **Logging**

Logging is a crucial part of any application, especially web applications, which run continuously. Logging allows programmers to see detailed records of a program's execution path. This can be useful in the event of an application crash or other problem. While a friendly error message can be displayed to a user, the full details of the error can be logged for a programmer to go look at at a later time. Logging, when used correctly,

can provide a useful debugging method for developers.

During the development of enDx, logging was built in to the application. It seems that enDx was initially written using two logging facilities: `GenericServlet.log` and `Logger`.

`GenericServlet.log` is a simple logger provided by J2EE for use in servlets that derive from `GenericServlet`. `GenericServlet.log` provides a log file for each servlet to which arbitrary messages can be written. There is no formatting imposed on the log messages, so `GenericServlet.log` is similar to `System.out.println`.

`Logger` is a logging facility that was written by the company that originally wrote enDx. Each instance of `Logger` writes all of its output to a single log file. `Logger` writes all of its logs using a fixed format that includes the severity of the message, the date and time, the amount of time the application has been running, the name of the currently running thread, and the message. `Logger` is also capable of logging messages at different severities, though it has no way to filter messages based on their severity.

As the application has grown, two new ways of logging have been added to enDx: `System.out.println` and `SingletonDebug`. As the name suggests, `System.out.println` is just the method from the J2SE libraries being used to log messages to the console, where the application server picks them up and puts them into the server's log file. `SingletonDebug` is used in concert with `System.out.println` to control logging. It is used as a simple code guard around `System.out.println` calls so that they can be disabled or enabled based on whether `SingletonDebug` is enabled in the application's configuration.

Having this disparate set of loggers has created some problems. First, it was not clear when to use each logger. With the exception of `GenericServlet.log`, the loggers could be used anywhere. This led to the loggers being used interchangeably in some parts of the code. The second problem this created was that the logs were split between different places. `Logger` could have any number of log files anywhere on the hard drive.

`GenericServlet.log` had a log for every servlet. `System.out.println` had its messages buried

in the server log file, which also contains log messages from the server and every other application running on the server.

To fix this issue, it was decided to standardize on a single logging facility. The Apache commons-logging API and the log4j logging engine were selected for this purpose in enDx. Commons-logging was chosen because it provides a common API for logging that has adapters for many common logging frameworks, such as log4j and Java Logging. The log4j package was chosen for the logging engine because it is supported by commons-logging, because it is well supported by the community, and because it is very flexible in what it can do.

The transition to the new logging system will be two distinct steps: setting up log4j and converting all logging calls to commons-logging. To set up log4j, a configuration file must be created that details the format that the log messages will have as well as where to write the log messages. The flexibility of log4j allows an application to write log messages to a variety of appenders, including the console, a file, a system log service or daemon, and email. Log messages can be filtered based on their severity, so one could, for example, have any fatal log messages be sent out as emails, while all other messages are logged to a file. A fairly simple configuration as chosen, where all messages of info severity or higher will be logged to both the server log and to a separate, application-specific log file.

The process of converting all logging calls to use commons-logging began by extracting the log message from the old log facility. The log message was sanitized to remove information such as severity, line number, and source file, since this information can be generated by the logging facility or set in the logging function call. After this, private, static loggers were created for each class. These loggers were used to log messages to the log engine, which could then determine what should be done with each log message using the configuration file.

Since the log messages that are at the trace and debug level are of greater detail than is needed for normal operation, they are surrounded by code guards. These guards

test to see if the log level (debug or trace) is enabled for output to one of the appenders, and, if one of them is, then executes the log statement. While log4j will only output log messages that pass through one of the filters, it still must parse the log function call, including the string concatenation that is most likely part of it. By using code guards, one can prevent the overhead of string concatenations and other method invocations that are part of building the log message when the log message does not need to be output.

Once this conversion is complete, logging should be much easier for developers of enDx. First, it will be easier for developers since there will be a single, standard, well-defined logging API that they will need to know instead of four that are not used anywhere else. Second, this API will be available in any application they work on since it has been made part of the MDEQ standard toolset, so they will be able to take their knowledge from other projects and more easily apply it to enDx. Third, it will provide a single engine to log through, making it easy to configure logging.

## IV. CONCLUSIONS AND RECOMMENDATIONS

It was not possible to complete enDx within the time frame of this thesis. The database schema has been altered and is currently undergoing testing. The application has seen some work, but none of the problem areas have been completely addressed. Work has been done in these areas to develop a common strategy to fix these problems so that the rest of the process can be as mechanical as possible.

### **Conclusions**

Currently, enDx is on the road to completion. A solid base of work for fixing the existing issues of enDx has been laid and is currently being enhanced by a slowly growing team. This thesis has detailed the work that has happened so far as well as work that will need to happen in the near future. Once the work detailed in this thesis is complete, enDx will be ready for waste tire functionality to be added.

### **Issues Encountered**

Many issues were encountered during the refactoring of enDx. First, there were no official testing procedures or unit tests. This meant that, when a change was made, it was nearly impossible to check to see if the change fixed an issue and did not cause a regression without pulling someone away from their normal job to test it themselves.

Second, only one person in the office knew the application well enough to test it. Not only did this mean that this person had to be constantly interrupted to test a new change, but that when this person was out of the office, it was impossible to test the application past basic functionality. This was a very large problem since the person in

question was out of the office about half of the time. This meant that, when he was able to test changes, he was testing so many changes that it was very difficult to tell which change had caused an issue to arise.

Third, we did not have the planned staff levels. The project was planned to be a software team management project, in which the author handled design and team management, and delegated most of the programming to members of the team. By the time the project had started, one of the team members had been reassigned to another division and the remainder of the team had been reassigned to other projects that had arisen after the project plan was assembled. The author was also given other projects to work on in addition to enDx, reducing the time available to work on this project. This led to the entirety of the project being done by the author. Because of this, instead of being half way to completion of implementing waste tire permitting and reporting, as was laid out in the original project schedule, the author is still refactoring the codebase in preparation for implementing waste tire permitting and reporting.

### **Recommendations**

The path that enDx is currently on should provide MDEQ with a solid version of enDx on which waste tire permitting and reporting can be built. While database work is mostly completed for the new enDx, there is still much work to be done in the application. Clean up of the JSPs is still needed in order to separate the controller and the view, as well as convert the application to use commons-logging.

One additional step should be taken. Due to the issues that have been caused by using Java stored procedures, MDEQ should migrate the code from stored procedures in the database into Java classes in the application. This should not be very difficult, since the only change that should have to be made is to change the way the code connects to the database. So, instead of using the static database connection that Oracle provides, the code will have to get a connection from the database connection factory that is part of enDx.

## **APPENDICES**

**APPENDIX A**

**ABET PROGRAM OUTCOMES**

**COMPUTER SCIENCE**

The Computer Science department at Kettering University set forth these four program objectives:

- 1 Computer Science graduates will have a broad, mathematically rigorous program in the fundamental areas of computer science that will allow them to continue their professional development and sustain a life-long career in computer science either through graduate study or continuing self-directed learning and development activities.

What I have learned at Kettering University has helped me to solve problems in the workplace using not only the tools that I gained in the classroom, but also by helping me learn how to find the answers myself.

- 2 Computer Science graduates will have developed a sufficient depth of understanding in computer science, and the skill, confidence, professionalism, and experience necessary for successful careers in computer science and related fields.

The knowledge and understanding that I gathered in the classroom at Kettering University has given me confidence in both the classroom and the workplace that has allowed me to be successful.

- 3 Computer Science graduates will have the teamwork, communication, and interpersonal skills to enable them to work efficiently with interdisciplinary teams in industry, government, and academia.

Through my classwork at Kettering University, as well as through my time spent with my employer, I have learned how to communicate effectively with my peers to help achieve things greater than I could have achieved on my own.

- 4 The Computer Science faculty will provide its degree majors an excellent education experience through the incorporation of current pedagogical techniques, understanding the contemporary trends in research and technology, and hands-on laboratory experiences that enhance the educational experience.

My time spent in the classroom and in the lab at Kettering University helped me learn about the field of computer science and gave me the desire and confidence to pursue a master's degree.

**APPENDIX B**

**ABET PROGRAM OUTCOMES  
COMPUTER ENGINEERING**

Each graduate of the Computer Engineering program will have demonstrated the ability to do each of the following:

*Program Outcome 1. Assembly language.* Analyze, design, develop, debug, and document structured assembly language programs for at least two different embedded-computer platforms, including at least one with a 32- or 64-bit architecture. Use appropriate techniques and modern embedded-computer development tools.

In my classwork at Kettering University, I developed programs for the Freescale 68HC11 (16 bit), Freescale HCS12 (16 bit), and MIPS (32 bit) processors in each processor's assembly language.

*Program Outcome 2. High-level language.* Analyze, design, develop, debug, and document programs in at least one structured high-level programming language. Use appropriate techniques and modern software development tools.

In my classwork at Kettering University, I had many opportunities to develop programs in high-level languages, such as C and Java.

*Program Outcome 3. Real-time operating systems.* Develop, debug, and document a simple real-time operating system and design, develop, debug, and document application programs for it to implement a complete real-time system that meets specifications. Use appropriate techniques and modern embedded-computer development tools.

In my real-time operating systems course at Kettering University, I was part of a team that developed a simple real-time operating system that was capable of running a weather station that monitored wind speed, maximum wind speed, and wind direction and displayed the data using LEDs, a seven-segment display, and a terminal.

*Program Outcome 4.* Analyze, design, prototype, debug, and document combinational and sequential digital circuits. Use appropriate techniques and modern digital-systems development tools and implementation technologies.

In my digital systems classes at Kettering University, I designed and implemented combinational and sequential logic circuits, including programming GAL devices to simplify circuit design.

*Program Outcome 5. Computer architecture.* Design and verify the operation of a basic central processing unit for a general-purpose computer. Use appropriate techniques and modern digital-systems simulation tools.

In my computer architecture course at Kettering University, I designed and implemented a simple pipelined CPU in a simulator.

*Program Outcome 6. Circuits, electronics, and systems.* Model, analyze (at DC and AC steady state), and design electrical circuits and systems. Use modern electronic design and test equipment.

In my circuits and electronics classes at Kettering University, I analyzed and designed circuits consisting of resistors, capacitors, diodes, and transistors.

*Program Outcome 7. Elective areas.* Use understanding of basic principals and appropriate tools to analyze, design, develop, debug, and document simple systems in at least two of the following areas of computer engineering: computer networks, programmable logic controllers, expert systems, database systems, VLSI systems.

Through my elective studies at Kettering University, I was able to gain knowledge in the areas of computer networks and database systems.

*Program Outcome 8. Teamwork.* Work productively in a multidisciplinary team, in particular to carry out projects involving computer engineering.

Throughout my coursework at Kettering University, I have participated in multiple teams to complete assignments, the biggest being a 5-person team for my capstone project.

*Program Outcome 9. Ethics and professionalism.* Act in a professional and ethical manner in the workplace.

Through my senior seminar in leadership and ethics at Kettering University, I learned to act in a professional and ethical manner.

*Program Outcome 10. Written and oral communication.* Communicate effectively through written reports and oral presentations appropriate for other computer engineers or for non-technical audiences, as required.

Through my communications courses at Kettering University, I learned how to effectively communicate my thoughts and ideas to other individuals, even when they are technical in nature.

*Program Outcome 11. Global and societal context.* Understand the impact of engineering solutions in a global and societal context.

Through my senior seminar in leadership and ethics at Kettering University, I learned how a poor engineering solution can impact society in negative ways.

*Program Outcome 12. Lifelong learning.* Independently acquire the information and understanding necessary to complete projects or undertake other responsibilities in unfamiliar areas from appropriate sources such as books, training courses, technical documentation, and application notes.

Through my time at Kettering University as well as my time with my employer, I learned how to find the information that I needed so that I could learn what I needed to know so I could get the job done.

*Program Outcome 13. Contemporary issues.* Understand contemporary issues, especially as they relate to employment as a computer engineer.

In my time at Kettering University and with my employer, I have been introduced to many contemporary issues and have actively pursued greater knowledge of them.