Propel Designer

Final Report for CS39440 Major Project

Author: Steven James Meyer (svm9@aber.ac.uk)
Supervisor: Prof. Reyer Zwiggelaar (rrz@aber.ac.uk)

April 12, 2013
Version: 1.0 (Draft)

This report was submitted as partial fulfilment of a MEng degree in Software Engineering (G401)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

In signing below, I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
- I understand and agree to abide by the University's regulations governing these issues.

Signature ....

Date ....

Consent to share this work

In signing below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Signature ....

Date ....

Acknowledgements

I am grateful to...
I’d like to thank...

Abstract

Databases can be usefully modelled as Entity Relationship Diagrams (ERDs). Such diagrams use visuals to show information about database schemata such as relationships and cardinalities, entities and attributes in a way which is easy for a human to recognise. Colouring and emphasis can further assist in using ERDs as a useful tool for sharing and communicating a database schema.

Propel is an open-source Object-Relational Mapping (ORM) for PHP. It creates objects to allow for programmatic database manipulation and for database abstraction. It creates these objects and the database implementation from schema files. These schemata define the tables, columns, keys, indices and relations and their attributes within a database.

Schema files are coded in XML. While this makes them understandable so far as the data is marked-up in a human-readable manner, it makes sharing and collaborating difficult. The linear structure makes it difficult to see the structure and relationships. This, in turn, makes it difficult to ensure that the schema is modelling the domain and following business logic correctly. They are also not very friendly for the less technically-inclined.

It can be seen, then, that it would be useful to have a tool which can be used to manipulate Propel schemata by means of direct XML manipulation and by ERDs.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Background & Objectives

1.1 Background

Data modelling and database design are vast topic areas accompanied by many volumes of published theory and literature. This project focusses on a fragment of the topic; conceptual design/ schemata. Specifically, it is concerned with entity relationship (ER) grammar and mainly with ER diagrams.

The ER modelling grammar for conceptual modelling serves two major purposes. Firstly, it can be used as a communication device used by an analyst to interact with an end-user. Secondly, it can be used as a design tool at the highest level of abstraction to communicate a deeper understanding to the database designer (Umanath & Scamell (2007)). Id est the database structure and semantics can be described in a manner which is not specific to any implementation.

Where object relational mapping (ORM) is used, such diagrams can also be useful devices for describing the relationships between objects to the application programmers. This is particularly useful where business logic is applied at the application level; developers (who may not have designed the database, themselves) must have a way to understand the logic represented by the database design and to understand those parts of the business logic which must be applied at the application level. This is particularly relevant when using an ORM such as Propel, where the level of database abstraction afforded to the application programmers comes at the price of losing the ability to describe logic at the database level. Database-level logic, in this case, is represented by stored procedures, triggers et cetera.

For the ORM Propel, database schemata are described as XML files. These descriptions are a mixture of database instruction such as entities (tables), attributes (columns) and simple referential actions for deletion and updating of rows, and of PHP instructions for the generated PHP code.

These files make for more difficult traversal for application programmers and database engineering alike, as references must be searched for among the (possibly) many entity descriptions and indices are separately described from the attributes they reference. It is also far too tempting for application programmers to make changes to the schemata when requirements change. Maintaining any ER diagrams in such situations is a maintenance issue, and checking that the resultant schemata still correctly and properly represent the business logic becomes very difficult.

There exists many database modelling tools which support the creation and editing of ERDs. Many of these tools are useful tools for inspecting existing databases, providing support for many different back-ends. However, they are primarily aimed at SQL interaction with these databases, rather than at maintaining ER diagrams.

There are products which will natively support ER diagrams and ORM frameworks, but these products are not free software. Existing software, as it stands, usually has one of these weaknesses:

- They do not support ORM. They support different databases, but cannot be made to work with an ORM schema.
- They do not support ERDs or relational modelling schema (a form of physical data modelling).
- They are not free software. They are either not *gratis* (without cost) or not *libre* (without restrictions) or both.

This project, rather than being of commercial importance or even being a common problem in industry, came about by the student's own need. Such a situation, however, is the precursor to every good work of software (Raymond (2002)).

1.2 Objectives

The most basic aim of this project is to build a free tool to create ER/RM diagrams from Propel schema XML and vice versa. It should provide the expected features of existing modelling tools, but be specifically aimed at Propel schemata. The normalised information-preserving grammar of Propel is more closely related to relational modelling grammar, so RM modelling grammar would be preferable to ER modelling grammar.

The core goals of the project are:

- To produce an schema diagram consisting of (*i*) tables, (*ii*) relationships (*iii*) attributes (and their types) and (*iv*) cardinality notations using a Propel schema as input.
- To make such a diagram interactive/editable with a database modelling tool.
- To produce a Propel schema XML using a schema diagram as input, id est the reverse of goal 1.
- To be gratis to use.

The project had these stretch goals which were available if the core goals were to be achieved in good time:

- Further integration with Propel to allow for additional logic such as validators and behaviours.
- Include inheritance. This is part of the Propel schema and introduces aspects of enhanced ER modelling.
- To be portable. These tasks do not require any OS interaction and so should not need installing, neither should any special software be required to use it.
- To be free software, id est libre.

1.3 Related Systems

As previously discussed, there are many existing 'database designers' available. One such product which has native support for ORMs, including Propel, is ORM Designer. However, this product is closed-source and costs €99 for a single licence.

The initial design for this project was to extend an existing software to include support for Propel schema. Of the various free modelling tools which were investigated, WWWSQLDesigner by Ondrej Zara was selected to be extended. This designer produces its own XML from RM diagrams, and database-specific SQL is generated from this using XSL transforms. The software is written in JavaScript and HTML meaning that it should run on any platform with a web browser making it a portable application and fulfilling one of the goals of this project.

There are flaws in the software, however, which limited its usefulness for modelling Propel schema (or, indeed, any database) such as lack of composite key support and other incompatibilities with transforming the XML. It also used an in-house JavaScript library created by Zara which made modifying the code to add extra features a challenge. It also has no facility to save the diagram as an image.

After these issues were identified, it was decided that the modelling software would be created as part of this project to utilise modern web technology and avoid custom frameworks.

Chapter 2

Development Process

2.1 Process Methodology

Selection

A schema diagram consists of a few main parts: the tables; the columns; and relationships. These are hierarchical in nature — tables have columns, columns have attributes and relationships depend on tables most significantly and then on columns. When viewed in this manner, it can be seen that an evolutionary process model can be used to develop the system.

It was decided than an agile approach to development would be used. The developer was also the intended customer for this project, although the software would presumably be usable by others, so the developer team would be small and the customer dedicated, collaborative, and empowered. As the system is evolutionary in nature, the requirements can be detailed as and when they are required, rather than being described in their entirety from early in the project.

The development process was mostly based upon elements of extreme programming (XP), adopting those facets pertaining to fast, iterative programming such as “you ain’t gonna need it” (YAGNI) and test-driven development (TDD). The requirements and tasks were formed using XP’s notion of stories.

In using this development model, it should have been the case that after every iteration there would be a functional (if not feature-laden) product. This gave some protection against unforeseen changes or set-backs, assuring that there would be a product at every stage of the life-cycle. It would also deliver value early in the production.

Once the development methodology had been chosen the high-level requirements stories were described. These initial stories were very simplistic in their specification and encapsulated the main aspects of forming a diagram of an entity. Due to the use of XP, failure to implement any of these stories would not affect the releasable state of the program.

When the original modelling tool was abandoned, these same high-level stories were re-usable in specifying the development of the new modelling tool after having their context altered to fit.

Modification

Some XP components had to be modified to cater to the requirements of the project. As the project was being undertaken by a sole developer, the practice of pair programming could not be accomplished in the expected way. As pair programming is an essential device for detecting and correcting errors and code smells, it was adapted to have some impact with a sole developer: the code was inspected at a later date in order to identify issues which had been missed in the first instance.

The YAGNI process was also relaxed slightly in order to accommodate for patterns for which it was clear would be required at a later date, but which would be prohibited by an XP purist as such action would be deemed superfluous in the instance in which it was being created. This decision was intended to reduce the amount (and cost) of refactoring which would be required when the XP process would eventually call for it.

Requirements Specification

Stories were committed to story cards to be arranged on a board in the vicinity of the development environment, but for stories which had been broken down into detailed tasks were committed to !PROGRAM! tickets. This program allows for tickets to be assigned, have comments added, and ultimately closed as completed allowing the work-flow to be documented and visualised.

2.2 Tools

The first implementation of the software was written in XSLT and JavaScript — the same programming languages as the intended design tool, WWWSQLDesigner. The final implementation which included a designer used JavaScript, scalar vector graphics (SVG), and HTML5. These programming languages were selected because of their ubiquitous support on all major platforms; all of these are available in most major web browsers. It was decided that the JavaScript would be compiled from CoffeeScript. This is mainly due to CoffeeScript's standardised way of dealing with classes in JavaScript and because the compiled JavaScript is purportedly more readable than and executes as fast as or faster than equivalent JS written without CoffeeScript and which is also pretty-printed and passes through JSLint without generating warnings (or errors).

All development was performed using the Netbeans IDE from Oracle Corp. The release candidate version 7.3 was selected for its enhanced HTML5 development environment and the CoffeeScript plugin was used for developing and compiling CoffeeScript. Using this IDE allowed the project to be debugged in the IDE, negating the need for additional web-browser inspection tools.

Chapter 3

Design

You should concentrate on the more important aspects of the design. It is essential that an overview is presented before going into detail. As well as describing the design adopted it must also explain what other designs were considered and why they were rejected.

The design should describe what you expected to do, and might also explain areas that you had to revise after some investigation.

Typically, for an object-oriented design, the discussion will focus on the choice of objects and classes and the allocation of methods to classes. The use made of reusable components should be described and their source referenced. Particularly important decisions concerning data structures usually affect the architecture of a system and so should be described here.

How much material you include on detailed design and implementation will depend very much on the nature of the project. It should not be padded out. Think about the significant aspects of your system. For example, describe the design of the user interface if it is a critical aspect of your system, or provide detail about methods and data structures that are not trivial. Do not spend time on long lists of trivial items and repetitive descriptions. If in doubt about what is appropriate, speak to your supervisor.

3.1 Overall Architecture

3.2 Some detailed design

3.2.1 Even more detail

3.3 User Interface

3.4 Other relevant sections

Chapter 4

Implementation

The implementation should look at any issues you encountered as you tried to implement your design. During the work, you might have found that elements of your design were unnecessary or overly complex, perhaps third party libraries were available that simplified some of the functions that you intended to implement. If things were easier in some areas, then how did you adapt your project to take account of your findings?

It is more likely that things were more complex than you first thought. In particular, were there any problems or difficulties that you found during implementation that you had to address? Did such problems simply delay you or were they more significant? Your implementation might well be described in the same chapter as Problems (see below).

Chapter 5

Testing

Detailed descriptions of every test case are definitely not what is required here. What is important is to show that you adopted a sensible strategy that was, in principle, capable of testing the system adequately even if you did not have the time to test the system fully.

Have you tested your system on ’real users’? For example, if your system is supposed to solve a problem for a business, then it would be appropriate to present your approach to involve the users in the testing process and to record the results that you obtained. Depending on the level of detail, it is likely that you would put any detailed results in an appendix.

5.1 Overall Approach to Testing

5.2 Automated Testing

5.2.1 Unit Tests

5.2.2 User Interface Testing

5.2.3 Stress Testing

5.2.4 Other types of testing

5.3 Integration Testing

5.4 User Testing

Chapter 6

Evaluation

Examiners expect to find in your dissertation a section addressing such questions as:

- Were the requirements correctly identified?
- Were the design decisions correct?
- Could a more suitable set of tools have been chosen?
- How well did the software meet the needs of those who were expecting to use it?
- How well were any other project aims achieved?
- If you were starting again, what would you do differently?

Such material is regarded as an important part of the dissertation; it should demonstrate that you are capable not only of carrying out a piece of work but also of thinking critically about how you did it and how you might have done it better. This is seen as an important part of an honours degree.

There will be good things and room for improvement with any project. As you write this section, identify and discuss the parts of the work that went well and also consider ways in which the work could be improved.

The critical evaluation can sometimes be the weakest aspect of most project dissertations. We will discuss this in a future lecture and there are some additional points raised on the project website.

Appendices

Appendix A

Third-Party Code and Libraries

If you have made use of any third party code or software libraries, i.e. any code that you have not designed and written yourself, then you must include this appendix.

As has been said in lectures, it is acceptable and likely that you will make use of third-party code and software libraries. The key requirement is that we understand what is your original work and what work is based on that of other people.

Therefore, you need to clearly state what you have used and where the original material can be found. Also, if you have made any changes to the original versions, you must explain what you have changed.

Appendix B

Code samples

2.1 Random Number Generator

The Bayes Durham Shuffle ensures that the psuedo random numbers used in the simulation are further shuffled, ensuring minimal correlation between subsequent random outputs

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0 - EPS)

double ran2(long *idum)
{
  /*-----------------------------------------------------*/
  /* Minimum Standard Random Number Generator            */
  /* Taken from Numerical recipies in C                  */
  /* Based on Park and Miller with Bays Durham Shuffle */
  /* Coupled Schrage methods for extra periodicity       */
  /* Always call with negative number to initialise      */
  /*-----------------------------------------------------*/

  int j;
  long k;
  static long idum2=123456789;
  static long iy=0;
  static long iv[NTAB];
```

13 of 16

```
  double temp;

  if (*idum <=0)
  {
    if (-(*idum) < 1)
    {
      *idum = 1;
    }else
    {
      *idum = -(*idum);
    }
    idum2=(*idum);
    for (j=NTAB+7;j>=0;j--)
    {
      k = (*idum)/IQ1;
      *idum = IA1 *(*idum-k*IQ1) - IR1*k;
      if (*idum < 0)
      {
        *idum += IM1;
      }
      if (j < NTAB)
      {
        iv[j] = *idum;
      }
    }
    iy = iv[0];
  }
  k = (*idum)/IQ1;
  *idum = IA1*(*idum-k*IQ1) - IR1*k;
  if (*idum < 0)
  {
    *idum += IM1;
  }
  k = (idum2)/IQ2;
  idum2 = IA2*(idum2-k*IQ2) - IR2*k;
  if (idum2 < 0)
  {
    idum2 += IM2;
  }
  j = iy/NDIV;
  iy=iv[j] - idum2;
  iv[j] = *idum;
  if (iy < 1)
  {
    iy += IMM1;
  }
  if ((temp=AM*iy) > RNMX)
  {
```

14 of 16

```
    return RNMX;
  }else
  {
    return temp;
  }
}
```

15 of 16

Annotated Bibliography

Inc., AquaFold. 2012 (Nov.). *Aqua data studio features*. `http://www.aquafold.com/aquadatastudio_features.html`. Accessed 2012/11/19.

Raymond, Eric S. 2002 (Aug.). *The cathedral and the bazaar*. `http://www.catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/`. Accessed 2013/04/12.

Sebesta, Robert W. 2007. *Programming the world wide web*. fourth, international edn. Addison Wesley.

Sherratt, Edel. 2011 (Nov.). *Entity relationship translation*. `http://www.aber.ac.uk/~dcswww/Dept/Teaching/CourseNotes/current/CS27020/Lectures/ER-translation.pdf`. Accessed 2012/11/18.

Stevens, Perdita, & Pooley, Rob. 2006. *Using uml: Software engineering with objects and components*. second edn. Addison-Wesley.

Team, Propel. 2012a (Aug.). *Basic relationships*. `http://propelorm.org/documentation/04-relationships.html`. Accessed 2012/11/19.

Team, Propel. 2012b (Sept.). *Database schema*. `http://propelorm.org/documentation/04-relationships.html`. Accessed 2012/11/19.

Umanath, Narayan S. & Scamell, Richard W.2007. *Data modeling and database design*. Thomson Course Technology.

van der Vlist, Eric. 2002 (Jan.). *Xsltunit*. `http://xsltunit.org/0/2/`. Accessed 2013/02/02.

w3schools. *Xslt elements reference*. `http://www.w3schools.com/xsl/xsl_w3celementref.asp`. Accessed 2013/02/02.