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 22, 2013
Version: 1.0 (Draft)

This report was submitted as partial fulfilment of a MEng degree in
Software Engineering (G601)

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'd like to thank...

- My supervisor, Professor Reyer Zwiggelaar, for allowing me to create this project at my own pace and for encouraging me to do my best.
- My long-suffering partner, Aylin Alieva, for putting up with me throughout all of my years at the University, but particularly for these final months of dissertation work.
- The Father and the Son, for giving me strength.

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. It should not impose any restrictions such as trial periods or usage limits.

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 make use of only free and open standards. The software should not include any closed-source technology.
- 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.

ORM Designer has explicit support for Propel. It includes facilities to edit the Propel-specific properties of entities and attributes, model tree navigation, associations and schema import and export. It also has support for behaviours and validators.

ORM Designer is available on all major platforms and has a portable version available.

With its visual ERD interface and all of these other Propel-specific features, it is clear that ORM Designer is a very strong candidate for achieving the goals set out for this project. However, the main issues preventing it from being viable and the reason why this project could compete, is that is is a closed-source commercial product with a single user licence requiring €99.

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, WWW SQL Designer by Ondřej Žára 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 Žára 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.

While WWW SQL Designer may have been dropped from the project, its merits were still such that it was used as a comparible project. This project aimed to integrate the best features of WWW SQL Designer and ORM Designer. It should be noted that ORM Designer's 14-day trial period has the implication that its 'best features' are those which can be gleaned from its own sales pitch and community reviews.

Chapter 2

Development Process

2.1 Process Methodology

2.1.1 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.

2.1.2 Modifications

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.

As this is a personal project there is no separate customer entity. This has its benefits in that the 'customer' will always be available, on-site, knowledgeable and authoritative; essentially everything a good on-site customer should be.

It also has its risks. As the customer is also the whole developer team, there is an introduced risk that the stories/requirements may not be recorded in the same or sufficient detail as if the stories were to be communicated to different people. In order to mitigate this risk, the requirements were shown to another person who was not involved in the project to ensure that the details were sufficient.

2.1.3 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 tickets using Trac. 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 Planning

2.2.1 Release Planning

For the iterative releases, stories describing similar functions were grouped together. These were loosely based around the different elements of the Propel schema: tables, rows and indices.

It was known that in the time which remained after the original plan was abandoned that it should have been possible to complete one release iteration. This release would incorporate the stories relating to loading, manipulating and saving schemata containing tables and columns and was expected to complete in 8 weeks.

This estimate was based upon the time which had been spent on the initial project, as this was the only planning experience available from which to plan. This release plan was better organised than the previous attempt, which had mostly descended into slightly-organised hacking.

2.2.2 Iteration Planning

As it was not yet known the velocity at which the project could sustainably operate, the tasks for the first release were divided up into 8 divisions of estimated equal work. While this is not how XP would divide up work, it was an exercise to discover just how much work could be undertaken in one day and in one week.

This gave a better gauge by which to estimate subsequent iterations. Tasks had different weights with regards to the work involved and so iterations were not equal in time, but they eventually stabilised to provide a constant flow of work.

Iterations were approximately one week long due to the small team involved, with tasks generally being possible to complete usually in one working day. These days were not in line with normal working days, as some days had to be committed to other task not related to the project.

2.3 Tools

The first implementation of the software was written in XSLT and JavaScript — the same programming languages as the intended design tool, WWW SQL Designer. 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).

The tests for the TDD process where created using unit testing frameworks for the programming language being used. For the initial development iteration being integrated with WWW SQL Designer, the XSLT style-sheets were tested by creating unit tests to be used with XSLTunit. This framework used XSLT documents as tests applied on the XSLT document to be tested in order to produce results as XML. The transformations are initiated by a web page using JavaScript, but the tests could be carried using any engine which can apply XSL transforms.

The CoffeeScript was tested using JsUnit, running the tests on the JavaScript produced by compiling the CoffeeScript into JavaScript. JsUnit tests are composed in JavaScript and the tests are run and aggregated by an HTML web page which displays the results and gives details of the errors.

Continuous Integration suites were not used due to the size of the project and the effort required to integrate a CI suite. The benefits which using such a suite would have brought to the project were emulated to some extent by running the suite of tests whenever the code-base was altered and when new tests were created. This ensured that modifications and new code did not affect the working state of other code in the project.

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 plug-in 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

3.1 Overall Architecture

The system was to be produced using agile/XP principles, so it should have been the case that the ideas of emergent design would give rise to a coherent design and good software as the project developed. Even with this principle in effect, it was clear that there was an overall design inherent to the project which separated it into distinct parts. These parts are:

- The database model.
- The diagram modelling tool and its interface.
- Code to link the previous two parts and to also transform the model into XML.

With these parts, it could be seen that the design pattern model-view-controller (MVC) would be a suitable pattern for the overall design.

As the entire software was to be run on a single machine, all of the programming was designed with JavaScript and its merits and limitations in mind.

3.2 Stories

The model, view and controller parts were not created as separate entities in isolation, as may be the case with a plan-driven process, but they evolved together over time as stories were implemented.

A story would describe a design so broad as "As a user, I need to add tables to a database so that my database has entities." This would then be separated into its tasks for the model and for the view. These would be tasks such as "A database should maintain a collection of tables" and "A database should allow tables to be added to its collection", which would both be tasks for the model. The related tasks for the view would be similar; "It should be possible to add a table to the database".

Following the principles of test-driven development, these tasks were used to create a black-box style test. Id est an expected outcome would be described and the test was considered satisfied provided that it passes, regardless of how the code accomplished the task. For example, a test would expect the outcome that the SVG has a new *rect* (rectangle) element of the class 'rect' after adding a table; how that came to be was not significant.

3.3 Coding Style

3.3.1 OOP and JavaScript

The model was designed to be entirely object-oriented. JavaScript is not a class-based language as Java is and, unlike most OO languages, has a variety of patterns for supporting OO programming styles.

CoffeeScript provides a class structure in order to simplify attaching functions to the prototype chain and also correctly handles the setting of the superclass. CoffeeScript uses the pseudo-classical pattern. In this pattern, objects are created with the `new` keyword and a constructor function — similar to conventional OO languages — and (public) methods are attached to the object prototype.

The prototype is used when member isn't found on the object itself. For example, if the object `foo` has no member `foo.bar`, then the engine looks for the object's constructor's prototype. This prototype object is inherited and, when used in this way, conserves memory as the members do not need to be copied to each instance.

Prototype members are *live* members; changes to the prototype will affect all current and future instances of the object. As a result, it is not a useful pattern for private members. The conventional handling of private members for this pattern is to prepend their names with an underscore. While this is useful for creating 'protected' members (JavaScript has no support for true protected members), it still exposes the member publicly as both readable and writeable.

In order to handle private members, it was decided that the all-in-one constructor pattern would also be used, but only for this purpose. This pattern adds all members to the object in the constructor and doesn't make use of the prototype at all. This is not entirely desirable, as inheritance breaks down somewhat (the `instanceof` keyword doesn't work, here) and every instance carries all of its members (no shared prototype object). It does encapsulate private members, though, and the pattern allows for the creation of 'privileged' members; publicly accessible members which have access to private members.

It was decided that all JavaScript objects would follow these rules:

- Private members would be defined in the constructor (all-in-one constructor pattern).
- Public members requiring access to private members (privileged members) would be defined in the constructor (all-in-one constructor pattern).
- Public members which are not privileged would be attached to the prototype (classical pattern).

3.3.2 CoffeeScript

Functions in CoffeeScript are defined by the means of

```
variable = (param1, param2, ..., paramn) -> function here
```

Where a function has no specified parameters, it is permitted in CoffeeScript to omit the brackets and start the function with the arrow (`->`). However, it was decided that functions without specified parameters would always include the empty brackets in order to clarify that the function is intended to have no parameters. It also serves an æsthetic function, causing it to become easier to identify functions.

3.4 Model

3.4.1 Structure

The model was designed to closely model the Propel schema reference. The structure of the XML schema document was to be represented in the model. As and when the agile method required them, this lead to the creation of 6 objects: Database, Table, Column, Foreign Key, Index and Unique Index. Foreign Key, Index and Unique Index were consigned to iterations beyond the time-frame available to the project as modelling tables and columns were deemed to deliver the greatest value.

The schema hierarchy has columns contained within tables, which, in turn, are contained within a database element. This becomes problematic when one considers that each element can inherit attributes from its container. As tables are referenced from the database in the model (Database "has a" Table), the table would require a reference to the database object for its values. This approach would have introduced tight coupling into the software which was rejected as bad practice. In order to flag those values which were to be inherited to differentiate them from those with their own values, a static-like variable was introduced to the models. Values which were to be inherited from the container were set thus:

```
table.setPhpNamingMethod Table.INHERIT
```

As the output XML would not require the true value, the inheritance would not need to be resolved by the software.

3.4.2 Visibility

As discussed in the previous section (3.3), JavaScript objects were created using a mix of the pseudo-classical and the all-in-one constructor patterns. This decision was particularly relevant to the objects in the model, as their attributes could either be public or private. Choosing one of these visibilities would change the pattern required for their access.

It was decided that attributes and collections would be private instance variables accessed by means of public get and set functions. This decision was made in the name of good practice. By mandating the use of these functions, any changes to the way in which attributes are get or set internally of the object would not require a change in the code which tries to get or set these attributes. As these functions are privileged, they follow the all-in-one constructor pattern.

By having the attributes and collections as publicly accessible properties of the objects, the code would have been easier to read and shorter, but it was rejected for the reasons already given and also for the reason that it would expose the internal data types, which are also liable to change.

3.4.3 Programming Language

JavaScript was selected as the programming language of choice by the merit of its inclusion on most platforms, devices and web browsers. Such penetration could not be matched by any other language available; Java requires a JVM installed and native programs are inherently restricted to their native platform. It also ties in conveniently with the other technology in use for the view and controller — that of HTML5 and SVG. All three are available together in most modern web browsers.

3.5 View

While researching other database modelling software, it was discovered that an oft lacking feature in demand in the forums was the ability to save the diagram as an image. With this in mind, the decision was made to have the diagram as an image, rather than using some other display technology such as HTML/HTML5 canvas or Adobe Flash. By using an image as the view, it would remove the need to convert the model into an image at a later stage and all of the complexities and potential incompatibilities which would accompany such a choice. That which was visible in the view would be exactly that which was in any exported image.

3.5.1 SVG

Scalar vector graphics (SVG) are images created by XML. This means that they can be created and interacted with programmatically by means of manipulation of the SVG's DOM. Being a vector image format also gives any exported image can be scaled without any loss of quality, unlike rasterised bitmap images which are limited in their resolution.

SVG are supported by all major browsers in their modern versions. This has the implication that no extra software or plug-ins are required to use the software, unlike other rich technology such as Flash or Microsoft Silverlight.

As the file is of an XML format, the semantics of the diagram and interface can be marked-up with ARIA attributes to improve the accessibility. This is unlike other RIA programs which may not have the same accessibility support as a web browser does.

SVG suffers from rendering speed degradation as DOM complexity increases. As this project will likely only be drawing a limited quantity of rectangles, text and lines per table, was deemed unlikely that this would cause problems for anything but very large schemata.

It was decided that jQuery would be used to interact with the SVG. Some of the jQuery library's function do not support SVG as completely as they do HTML, but these are clearly noted in the jQuery documentation and the work-arounds, where required, should not require a great deal of JavaScript.

3.5.2 HTML5 Canvas and HTML

HTML5 canvas was considered due its JavaScript API and its standalone support in major browsers. However, it was quickly rejected as it lacks the capability for dynamic components. In SVG the individual entities can be interacted with whereas the entire HTML5 canvas must be redrawn in order to alter it. Such behaviour would degrade the user experience for schemata with more than a few tables, as larger and larger canvasses (images) would need to be redrawn every time something is added, removed or moved as the number of tables increases.

HTML was considered for its superior support and superior accessibility, but the advantages of SVG caused HTML to be rejected. HTML would also require an interpreter to create an image of the diagrams.

3.6 Controller and User Interface

The controller was designed to maintain the JavaScript model, ensure the integrity of the SVG view, and to handle the interaction between the two.

The controller was designed to be in two main parts: a jQuery plugin to interact with the view and an HTML5/Twitter Bootstrap API to allow the user to change the model and view. The latter

consists of an HTML5 interface from which the user can add tables and such with some JavaScript to control the model; the interface delegates view interaction to the jQuery plugin.

In order that this interface be testable, the components of the Twitter Bootstrap framework were created as JavaScript objects which extend the jQuery object. Having the components as controllable objects instead of simple HTML to be inserted into the DOM allowed the interface to be constructed in such a way that it was predictable, testable and much simpler to read in the code. Rather than constructing many snippets of HTML, one can create an object and expect certain functions to be present. These functions replace the many jQuery functions required to do the same task with one function call. Having them extend jQuery enables them to function as jQuery object, with all of the functionality that jQuery allows.

Using the Twitter Bootstrap framework also presents the user with a familiar interface as the same framework is becoming more popular on the world wide web. It is also well tested and polished, reducing the work required for the user interface in this project.

Another key reason for using jQuery and Twitter Bootstrap is in the fact that both are designed to work well and consistently in all browsers and on all devices. This is important as it reduces the amount of work which would otherwise be required to work around the major and minor nuances of the different JavaScript engines, rendering engines and platforms, such as the notoriously unwieldy Internet Explorer 7.

3.7 Other Technology

3.7.1 Adobe Flash

Adobe Flash has a scripting API available and is capable of performing as a user interface for the software. It is not a native part of a web browser or system, but is often present on many systems due to its pervasive usage for both local content and on the world wide web. However, Flash is in decline on mobile devices and Adobe has discontinued its support for mobile platforms. While this was not a reason to reject it (this project is not aimed at mobile support, yet), it sets a precedent for Flash's future as developers will not be too happy to produce desktop content in Flash and mobile content in HTML5.

Flash is also not an open standard and this fact combined with the previous information and the fact that taking the time to learn Flash would not have been in the best interests of the project resulted in Flash not being considered for use.

3.7.2 Microsoft Silverlight

Microsoft Silverlight is an application framework for RIAs. While this project does not require an internet connection, Silverlight could still have been used to create the project. The interfaces of Silverlight applications have an API accessible by a subset of Microsoft's .NET framework.

Silverlight is available on Linux and FreeBSD by the use of the Moonlight project. However, the team responsible for the project — the Mono team — have abandoned its development. There are also issues with the licensing agreement between Microsoft and Novell and the product's penetration and use (or lack thereof).

Silverlight was rejected for similar reasons to that of Flash.

3.7.3 Java and JavaFX

Oracle has recently introduced JavaFX for creating user interfaces using Java for Java applications and RIAs. JavaFX would not have been an unreasonable choice for developing the model and the view as it could support all of the features required for the model, view and controller and handled the interactions between them. Java was rejected in favour of JavaScript, HTML5 and SVG as these are available without the need to install extra software on most platforms.

3.8 Other design decisions

3.8.1 Boilerplate

It was decided that the HTML5 boilerplate template generator *Initializr* would be used to generate the base code to start the view and controller. This generator creates the index.html page which includes some boilerplate code for displaying correct content in browsers and it also includes the Twitter Bootstrap CSS and JavaScript, the latest jQuery JavaScript, the Modernizr feature detection JavaScript and some extraneous pieces which are more useful for content which is to be served over the web.

Modernizr detects the HTML5 and CSS3 features which the browser supports. Initially, this is of limited use, but as the project progresses it may become useful to have feature detection. While this may seem at odds with the principles of agile development, Modernizr also includes scripts to add HTML5 functionality to browsers which don't support them natively and has been included for this reason.

Chapter 4

Implementation

4.1 WWW SQL Designer

4.1.1 XSLT

The test-driven development of the XSLT for the initial version of the software was reasonably straight-forward. The main issues arose from the lack of familiarity with XSL transforms, generally, as instruction for executing certain transforms frequently had to be looked-up in the early stages of the development.

One drawback with using the XSLTunit framework was in the engine's lack of support for importing tree fragments from external documents. This issue stemmed from the engine's poor implementation of the EXSLT (XSLT extension) for importing tree fragments, rather than an issue with the unit testing framework. In not having this extension available, it became the case that XML had to be embedded in every test which was specific to that test in which it was embedded. Having one block of test XML would have ensured consistent behaviour from the XSLT, rather than the emergent situation where the tests only proved their validity with the XML in isolation. It also increased the size of the tests.

The XSL, alone, could not model the Propel schema in its entirety. The WWW SQL Designer software was intended to be a generic schema designer for as many different database back-ends as possible. In this regard, it has the same aims as Propel which aims to provide a level of abstraction from the database back-ends. However, the Propel schema adds Propel-specific instructions/ attributes to the elements it describes, some of which do not have bearing on the databases, but should be included in the schema nonetheless.

4.1.2 oz.js Library

In order to extend the software to include this extra detail, the JavaScript and user interface needed to be altered.

The original author of the software, Ondřej Žára, has included a JavaScript library of his own creation, oz.js, to handle advanced JavaScript constructs such as custom events and class-like inheritance. This library is used extensively throughout the project. However, when examining the code in order to interact with the models, it was not possible to discover a method for extracting the models from the data structures in this library. Handles to the model object could be retrieved, but they were lacking the functions which seemed to be available inside of the library. This resulted in the models being uneditable without modifications to the library, which it was decided was

outside of the scope of the project and would have required much learning in order to understand the library.

A post on the Google Code mini-site for the project suggested that the WWW SQL Designer project make use of the jQuery library, as this library has a larger user base and is better documented. This would allow developers to contribute to the project without requiring them to first learn how to use the new library.

It was this difficulty and the previously discussed post which prompted the decision to abandon the WWW SQL Designer project and create a new system from the ground up using more popular frameworks and libraries. The move would allow the designer to be built with Propel in mind and to address other issues with Žára's project, such as the ability to save the schema as an image.

4.2 Propel Designer

4.2.1 CoffeeScript

CoffeeScript was a new programming language for the developer and, as a result, had to be learnt in order that it would be usable. However, as the body of the code is either JavaScript or code which compiles 1:1 into JavaScript, the learning curve is minimised greatly. There were occasions where the language was so different that the code had to be prototyped in the CoffeeScript real-time compiler on the CoffeeScript website in order to discover exactly how to structure the required JavaScript code as CoffeeScript.

CoffeeScript includes mechanisms to simplify OO programming and inheritance in JavaScript. The ability to set out a 'class' in CoffeeScript in a similar way as one would in a language such as Java was very useful, as traditional OO languages such as Java were much better understood.

Classical OOP in JavaScript is somewhat convoluted and its complexity often leaves it shunned for simpler patterns such as the prototypical Module pattern using functions and closures. Indeed, many libraries including jQuery use a prototypical `extend()` mechanism to emulate inheritance.

Having 'true' inheritance automatically applied by CoffeeScript brought all of the benefits of OOP to the project including polymorphism.

4.2.2 jQuery and SVG

jQuery was the library of choice for interacting with the DOM. This included the DOM fragment which was the SVG used for the view. jQuery is designed for the HTML DOM specifically, and this raised a few problems with the slightly different DOM of the SVG specification. These were mainly problems relating to jQuery's class methods as SVG elements maintain their classes differently to HTML elements. There are also some jQuery functions which do not exhibit the same behaviour with XML (and, by extension, SVG) as they do HTML. These were overcome by using workaround such as getting the ClassList property and performing standard JavaScript string functions upon it to imitate jQuery's class functions.

There exists jQuery plugins to address the issues with SVG, but these require a custom version of the jQuery library in order to make the plug-ins usable and so it was decided that the workarounds were a better compromise than a modified library, which may not be as up-to-date as the official libraries.

jQuery proved problematic with the production of the Bootstrap library. Functions which were desirable to override, such as `append()`, could not be overridden using the prototype as would be the case with OO programming. This was due to the way in which jQuery functions are applied.

As discussed with regards to CoffeeScript, jQuery does not use classical OOP. jQuery functions are instead called like a prototypical function, such as `$.append(content[,content])`, but this function does not exist in the object's prototype. It instead calls a function, `$.fn.append()`. This means that attempting to override this function by adding a function with the same name to the extending object's prototype still results in the jQuery function being called.

This was to be overcome by adding a function with the same name to the object (instance), instead. This is less efficient than using a prototype or extending jQuery as every instance maintains its own `this.append()` function. However, the functions attached to `this` are executed instead of the jQuery function.

4.2.3 Propel Inheritance

Determining a method by which to implement Propel's attribute inheritance from ancestor elements proved to be problematic as the inheritance direction is the opposite way around to that of the OO inheritance approach being used. A substantial amount of time was consumed in exploring this, and the outcome is described in the design section 3.4. As the Propel software determines inherited values at compile-time (when compiling the schema to produce PHP objects and SQL), this software does not need to concern itself with the actual values, but whether an attribute is defined or inherited is still a concern.

Methods to have the actual value of inherited values calculated during run-time were considered, but this was a case of over-thinking that which was actually required. Some time was lost to this and it was certainly not an XP process.

4.2.4 The User Interface

4.2.4.1 Twitter Bootstrap

When building the UI from snippets of jQuery and JavaScript, it was found that there was a considerable amount of repetition and duplication and should have required additional testing frameworks. Using the Twitter Bootstrap had saved a considerable amount of work with regard to style, but simple tasks such as alerts and changing styles where the cause of much duplication and causing the code to become difficult to read.

The UI, in its first incarnation, was mainly hacked together to display the correct buttons, dialogues and inputs. This approach quickly caused the code quality to deteriorate, introduced bugs and duplication and was ultimately unreliable and not testable.

A scripted version of the Bootstrap components could not be located, and the similar interfaces also called for bespoke reusable objects. Such objects could be tested using JavaScript testing frameworks which were already in place. The object created extend the jQuery object and can decorate jQuery objects to extend their functionality. This created some very powerful UI objects, with repetitive operations having their own functions such as

```
inputcontrolgroup.error("Oh, snap!")
```

which would handle changing the class to use the in-built Bootstrap style and also add the text to the expected place. By extending the jQuery object, a developer could interact with the HTML in the same way in which they would in normal circumstances.

Attempting to extend jQuery objects proved problematic as previously discussed. The library is not classically object oriented. This is why it is recommended to create jQuery plug-in to extend jQuery's functionality, rather than to extend jQuery objects. As the objects were mainly created

to reduced duplicated HTML and jQuery function calls, a plug-in was not suitable as plug-ins interact with existing jQuery objects; the Bootstrap library creates them.

A future release plan will combine the two approaches to allow for a jQuery plug-in interface which calls the Bootstrap objects prototype methods. This would be a more memory-efficient implementation and also provide for a more predictable flow of control.

While the development of the library added some useful and powerful features to the UI, it was also time-consuming to create and perhaps should be broken out into a project in its own right. It did allow for tests using the existing JsUnit framework and, as a result, a predictably scripted user interface.

4.2.4.2 File API

The decision to load and save directly from XML files on the host machine also created a substantial amount of work. This is due to JavaScript engines' paranoia with regards to file-system access from within JavaScript. These are security concerns relating to arbitrary file-system access from potentially malicious scripts over the internet. However, HTML5 introduced a File API which allows end-users to select files from their file-system with which a script can read from and, occasionally, write to files.

The File Reader API is a mostly event-driven interface. This initially caused some problems as it shared the XML loading code with the mechanism for loading XML from a text input, which is sequential. Refactoring the code to allow for asynchronous loading of XML was trivial, however, and did not upset the project's velocity.

The File Writer API was briefly examined and appears to have volatile support in view of its security implications. Writing to files did not fit into the final iteration, however.

Chapter 5

Testing

5.1 Overall Approach to Testing

This project was designed using a test-first design process. This process applied to every fragment of code in the project ergo it can be said with confidence that every line of code is tested, at least at the modular level. These tests validate that the code produces predictable objects and code.

5.1.1 Unit Tests

All of the tests in this project have been written in JavaScript as JsUnit unit-style tests. This has come to be as a result of the extensive use of CoffeeScript/JavaScript throughout the project. As the software is programmed in JavaScript, so the tests are written in JavaScript.

5.1.2 Acceptance Tests

Acceptance tests were created from the users stories selected at each iteration. These tests involved a real user enacting that which was described in the user story. For the most part, this would involve using the user interface to add a table to the schema and observing that a table with the correct attributes (name) was present on the screen, or a similar test.

Unit tests were present to check that such items were present in the DOM and visible, but this could only check that the test passed in a logical sense. The acceptance test would verify that the user could see the result, too, in an acceptable manner, rather than the unit test's assertion that it did exists somehow on the screen.

Such tests, however, could not be automated as they involved a human observer and human judgement. It would have been convenient to have these tests automated, however, as they could have been included as part of the automated testing suite and identify any errors introduced as the project developed.

In its current form, changes to the code base which effected the acceptance tests could only be detected by manually trying each acceptance test. This process was understandably slow, and prone to missing errors.

However, learning how to use and implement a new testing suite for these user interface tests (which most of the acceptance tests were) was deemed to expensive with the limited time remaining after the project change for the small quantity of acceptance tests which it would have automated. At some point in the future, when there are more acceptance tests, then a suite will inevitably be required and the tests retroactively codified.

5.1.2.1 Model Testing

The unit tests came about as implementation checks for the tasks which, themselves, were a breakdown of the design stories. In the nature of TDD, the tests would be written so that they would fail initially. The code was then created which would satisfy the test conditions. Although testing in this way can only prove that the implementation works as expected for the given test code, it is reasonable to expect that it works in the expected way for all cases.

The tests all consisted of unit tests written in JavaScript; the same programming language used to create the models (or, at least, the models in their compiled form). Unit tests were used extensively as the acceptance tests were mostly related to the user being able to perform tasks with the user interface, which would be interacting with the model. Having the model extensively tested with unit tests allowed for the assumption that acceptance tests on the UI would be adequate validity indicators.

Formal methods could have been used to completely verify and validate the design and the code, but such an undertaking would have required considerable time and effort and, as such, was not deemed appropriate for this project; unit testing provides adequate reassurance.

5.1.2.2 User Interface Testing

Rather than using a dedicated UI testing framework, the JsUnit framework was also used to test the UI. This was deemed an adequate testing solution as the UI was constructed from jQuery objects and the Twitter Bootstrap library. As the Bootstrap library was built, so unit tests had been created to validate the code. Bespoke components created from these library objects were also unit tested and so this gave reasonable assurance that the UI component would be created in a predictable manner.

In this way, it was possible to build the front-end without a user interface testing framework. Unit testing could not verify attributes such as the positioning of element in the web page, but the Twitter Bootstrap has extensively tested positioning and layout properties built in, so these were not a concern for this project. The style, layout and positioning of Twitter Bootstrap components is predictable and consistent.

Acceptance tests were bases around the black-box testing of the User Interface. These tests verified that the user could fulfil the stories and made it possible to say with confidence that the story had been completed, and also that the user interface was as complete as required by that same story.

5.2 Other Testing Methods

5.2.1 Automated Testing

The unit testing for this project is not automated. It was decided that the project was sufficiently small enough omit automatic testing.

To ensure that code changes had not affected the code in other parts of the system, the test suite was executed in full every time a new test was written (to prove that the new test failed) and when code was modified. These tests are sufficiently few in number than running all of them requires only a few seconds on the development machine.

All of the test groups can be run from the one test suite file, so there is some automation in that this one file will run all of the tests.

Were the project to grow in size and in the number of contributors, then automatic testing would be considered. The JsUnit framework includes files for integrating the test suite with continuous integration software.

5.2.2 Integration Testing

Integration testing has also not been used in this project. As the project's roadmap involves several modules working together, integration testing should be a vital tool for verifying the functional, reliability and performance metrics for the design.

Its exclusion from the project in its early development stages are because there aren't a great quantity of modules in existence at this stage. It was decided that implementing integration testing facilities at this stage would be too time consuming with regards to the given deadline.

The combination of error-free unit tests and a selection of acceptance tests which are also without errors gives reasonable assurance that the few existing parts of the software are working and are working together insofar as these few modules aren't conflicting. This is only a reasonable assumption while the project is so small.

It is expected that I&T will be in place for the next iteration in order to ensure that the different modules are still working coherently. Coherence can be tested very quickly manually in the project's current stage by running the program. As the project grows it will no longer be a feasible, reliable test.

5.2.3 User Testing

As the designer section of the project is designed to be interactive, user/usability testing is a useful tool to discover how users use the system and to discover how to improve the efficiency, accuracy, recall of completing tasks and the emotional response of the users.

As the project has not yet implemented many features and the amount of time which comprehensive testing would require, user testing has not been used in this project. It may, however, be used after more iterations, where greater value can be extracted from the investment in user testing.

5.2.4 Stress Testing

As the software is designed to be a single-user program and not expected to handle vast quantities of data, stress testing has not been used with this project. Using the program with the small test schema feels responsive on the developer's own machine and the time required to load the schema and the load on the CPU is minimal.

Perhaps it would be prudent to test the software with a very large schema to observe how it handles large amounts of data, but it could be argued that such a large database should be considered for splitting up into smaller schemata. Of course, the software does not yet handle imported schemata to form an aggregated schema. Such stress testing may be included in the project in the future, particularly when it handles more data per table than it does in its current, early incarnation.

Chapter 6

Evaluation

6.1 Scope & Design Decisions

The scope of the project, initially, would have been sufficient to have lead to the creation of a software which fulfilled all of the goals set out for the time-frame. Had the WWW SQL Designer project been simpler to understand in its libraries, then all of the diagram functionality would have already been in place, and would only have required extending to support Propel attributes.

The decision to create a new diagramming software instead of learning and modifying the in-house library was the correct one. While learning the library may have been less time consuming in the long run, by switching to a better known library, jQuery, this ensures that future engineers will be able to work more efficiently with the project.

Creating the diagramming software from the ground up also allows the software to be built around Propel and also to address the issues with the WWW SQL Designer project such as the lack of composite keys, printing and drag-and-drop from the outset, rather than trying to retrofit them into an existing system.

Creating a new library for the front-end elements may not have been a sensible decision with regards to the time allocated, but it does feel like a sensible decision with regards to the long-term development of this project. The repetitive nature of these elements would mean that the agile development process would have called for it in the very near future. It was the repetitive nature which prompted its creation, here.

The classes which the library has created are very powerful and useful. There are some issues with them as discussed in the Implementation chapter (4.2), but these should be easily resolvable. It should be noted that the library, like the rest of the project, is far from complete.

6.2 Requirements

The requirements for the project were effectively a summary of the elements which make up a Propel schema. This schema is reasonably well defined in its documentation and so, too, the requirements for this project were well defined.

However, when these requirements were extended or adapted for the new front-end, they were not so well defined. As a result of the project stemming from the developer's own needs, they were perhaps too focussed on that person's own expectations and so were less well planned and as comprehensive as they should have been.

6.2.1 Process Methodology

As the definitions of the Propel schema were already comprehensively described in Propel's own documentation and as the functions of a ER modeller are already fairly well established, it could be argued that extreme programming may not have been the best process methodology.

XP is better applied to cases of emergent design and where change is expected to be frequent. The almost static nature of the rules and the slow change cycles of Propel give weight to the argument that a less extreme agile approach or even a spiral-style life-cycle could be more appropriate.

However, it was the case that change still occurred frequently in this project, such as with the abandoning of the existing system and in creating a UI library. In this respect, an agile approach in which the requirements and time-scales could be changed inexpensively proved to be the best process.

6.3 Tools

The Netbeans IDE has been an indispensable tool in the making of this project. The most recent version at the time of writing (7.3 RC) has all of the facilities expected from an IDE for JavaScript and HTML5. However, most of the JavaScript created by this project was generated from CoffeeScript, which the Netbeans IDE supports only by means of a plug-in. This plug-in is missing syntax highlighting, but is otherwise useful for detecting mistakes and for compiling the CoffeeScript.

The IDE also has an integrated git version control client, but the Windows client from GitHub provides a more user-friendly interface and satisfying experience.

The new HTML5 additions to the Release Candidate also include JavaScript debugging and page inspection. However, these were not very easy to use. The developer tools integrated with Google Chrome and in the Firebug extension for Mozilla Firefox were much better tools.

The JavaScript unit testing framework, JsUnit, is and has been for some time an abandoned project. The team which was responsible for its development have moved on to a different JavaScript unit testing framework. However, the tool is still adept at testing JavaScript and presents it results in a useful way. The project did not try to use any of its integration facilities with automated regression testing suites.

It may be prudent to move the tests over to a framework which still enjoys active support, but as JsUnit has no issues, yet, this is not a major concern.

In all, the varied tool-set provided a suitable development environment, and did not hinder progress at any stage of the process.

6.4 Goals & Expectations

The project is not at a stage where it would be a useful tool for anyone and, for this reason, has not been released for anyone to use. However, the software is mainly a personal project and, in this capacity, I feel that the project is heading in the right directions and will, in a few more iterations, deliver the value which it was designed to.

The fact that there is so little useful functionality would suggest that, perhaps, the project was too ambitious to have been attempted within the time-frame. However, the project's initial goals were visualised at a time when the project was expected to extend the functionality of an existing open-source project. In this respect, the goals were entirely reasonable.

The expectations were also over-estimated. The amount of time which would be available for the project and also the amount of useful effort which could be afforded to the project were optimistic. In spite of this, the work which was completed was performed to a high standard and the project has created a solid foundation upon which to continue developing the software and create a useful software development tool and a powerful UI library.

The project is currently delivering on its extra goals of being portable and of using only free and open standards. It is composed entirely of HTML5, JavaScript and CSS. This technology is already available on most platforms, so the software can be run on most platforms. The software also does not need to be installed in order to run. HTML5, JS and CSS are free and open with many free and open-source engines available on which to run them.

The remaining goals are either longer-term design points (eg. more Propel integration and inheritance support) or goals which will be achieved when the software in in a usable — though not necessarily complete — state: being free, open-source software. When there is a more substantial starting point, the GitHub project will be made public and other developers will be encouraged to get involved to continue its development.

6.5 Reflection

As creating a library for Twitter Bootstrap was a major time sink, it may have been more efficient to use a front-end library which already exists, such as jQueryUI. This library achieves the same goals as the library created here, but using its own HTML elements, as opposed to Twitter Bootstrap components.

Bringing more developers on board earlier on, such as open-sourcing from the very beginning, is certainly a decision which may have lead to quicker development. However, due to the project's late start, gathering support and developers was considered a less efficient use of time than starting the project with a single developer.

Appendices

Appendix A

Third-Party Code and Libraries

1.1 jQuery

The jQuery JavaScript library is used extensively throughout the view and the controller sections of this project. It can clearly been seen in use as a function call in JavaScript called `jQuery()` or, more commonly, as `$()`. It is used wherever DOM traversal and manipulation and event handling are required. It is also used in those unit tests which require these functions.

This library was included in its original form as version 1.9.1, minified. It was bundled with other JavaScripts by Initializr. The original source can be obtained from `http://jquery.com/download/` or from `http://github.com/jquery/jquery`.

It resides in the file `public_html/js/vendor/jquery-1.9.1.min.js`. The development version is also in this location.

jQuery is licensed under the MIT licence.

1.2 Modernizr

The Modernizr JavaScript library is not utilised by any of the code created as part of this project. It is a self-contained JavaScript file which contains shims to implement HTML5 behaviour in browsers which do not support all of the HTML5 features. It is linked in by the projects only web page, `index.html`, by means of a script tag. It may enjoy use from other libraries, but none of those sources were created for this project.

This library was included in its original form as version 2.6.2, minified. It was bundled with other JavaScripts by Initializr. The original source can be obtained from `http://modernizr.com/download/` or from `https://github.com/Modernizr/Modernizr`.

It resides in the file

`public_html/js/vendor/modernizr-2.6.2-respond-1.1.0.min.js`

Modernizr is licensed under the MIT licence.

1.3 Twitter Bootstrap

The Twitter Bootstrap consists of two parts: the Bootstrap components and some jQuery plug-ins. The components are CSS styles, and these are included in the project as compiled and minified CSS files. These styles are used by means of class names in the HTML parts of the project and in the arrangement of mark-up.

The jQuery plug-ins included as part of the framework are:

- Transitions
- Modals
- Dropdowns
- Scrollspy
- Togglable tabs
- Tooltips
- Popovers
- Affix
- Alert messages
- Buttons
- Collapse
- Carousel
- Typeahead

Only Transitions, Modals, Dropdowns and Alert messages are currently used by the front-end and the Bootstrap API mini-project. All of the plug-ins are included by default by Initializr. They are all contained in one JavaScript file.

Twitter Bootstrap also includes a set of icon sprites by Glyphicons called Halflings. Some of these are used in the front-end in the Add, Save and Load buttons.

The CSS and bundled JavaScripts are included in their original form as version 2.3.0, minified. The source can be obtained from `http://twitter.github.io/bootstrap/assets/bootstrap.zip` or from `http://github.com/twitter/bootstrap`.

The CSS, JavaScript and sprites reside at

```
public_html/css/bootstrap-min.css
public_html/css/bootstrap-responsive.min.css
public_html/js/vendor/bootstrap.min.js
public_html/img/glyphicons-halflings.png
public_html/img/glyphicons-halflings-white.png
```

Twitter Bootstrap code is licensed under the Apache License v2.0.

1.4 JsUnit

JsUnit is the project name of multiple JavaScript unit testing frameworks. The project in use in, here, is by Pivotal/Edward Hieatt. The framework is no longer actively developed or supported, but the tests have been written and may in the future be transposed to another framework.

All JavaScript testing uses this framework. All content in the `test/jsunit/` location is unmodified, with the exception of the unit tests written for this dissertation project. These created

files are distinguished from the unit testing tests and example tests by their being suffixed with 'Test' or being a JavaScript file. These are in the `test/jsunit/tests` location.

The version included with the project is indeterminate and may not be the same version available at `https://github.com/pivotal/jsunit`.

JsUnit is licensed under the GNU GPL licence.

1.5 H5BP — HTML5 Boilerplate

The Initializr bundle has included some additional files and partial files. These are the remaining files in `public_html` which are not index.html and sample.xml. These other files are:

- `.htaccess`. A sample Apache configuration file. As this project is designed for a local machine, this file is not required or used.
- `404.html`. A boiler-plate File Not Found error page. This is designed for web servers, and is not used.
- Apple touch icons. These are used when the project is used on an Apple touch device.
- `favicon.ico`. This icon would be displayed in the title bar on a web page.
- `humans.txt`. A human-readable description of the website. It has not been populated, yet.
- `robots.txt`. A sample file to advise web robots on how to traverse the site. It is not used.

A boiler-plate index.html file was bundled with Initializr. This file has been modified for the project, but still contains the "Chrome Frame" and the JavaScript to load jQuery and the Twitter Bootstrap jQuery plug-ins. It also retains the `link` elements with link the Bootstrap stylesheets.

HTML5 Boilerplate is licensed under the MIT licence.

Appendix B

Code samples

2.1 CoffeeScript Class — Table

This class is an example of using CoffeeScript for OOP. This example contains the standard pseudo-classical pattern for public properties and the all-in-one constructor pattern for private and privileged properties.

This example has been subjected to line wrapping. Whitespace in CoffeeScript is significant and, unfortunately, the wrapping here would not constitute well-formed CoffeeScript code.

```
class Table
  @IdMethod =
    NATIVE:   "native"
    NONE:     "none"
    toArray: () ->
      this[method] for method of this when method isnt "toArray"

  @INHERIT = "__inherit__"

  @PhpNamingMethod =
    CLEAN:       "clean"
    NOCHANGE:    "nochange"
    PHPNAME:     "phpname"
    UNDERSCORE: "underscore"
    toArray:     () ->
      this[method] for method of this when method isnt "toArray"

  @TreeMode =
    MATERIALIZED_PATH: "materializedPath"
    NESTED_SET:        "nestedSet"
    toArray:           () ->
      this[mode] for mode of this when mode isnt "toArray"

  constructor: (name) ->
    _attributes =
      abstract: false
      allowPkInsert: false
```

```
    baseClass: Table.INHERIT
    basePeer: Table.INHERIT
    description: null
    heavyIndexing: Table.INHERIT
    idMethod: Table.INHERIT
    isCrossRef: false
    name: null
    namespace: Table.INHERIT
    package: Table.INHERIT
    phpName: null
    phpNamingMethod: Table.INHERIT
    readOnly: false
    reloadOnInsert: false
    reloadOnUpdate: false
    schema: Table.INHERIT
    skipSql: false
    treeMode: null
  _columns = new LinkedList()
  _foreignKeys = new LinkedList()
  _indices = new LinkedList()
  _uniqueIndices = new LinkedList()

  if not String::trim
    String::trim = () ->
      this.replace(/^\s+|\s+$/g, '')

  @addColumn = (column) ->
    if column instanceof Column
      _columns.addItem column
    else
      false

  @addColumnBefore = (column, before) ->
    if column instanceof Column
      _columns.addItem column, before
    else
      false

  @addForeignKey = (foreignKey) ->
    if foreignKey instanceof ForeignKey
      _foreignKeys.addItem(foreignKey)
    else
      false

  @addIndex = (index) ->
    if index instanceof Index
      _indices.addItem(index)
    else
```

```
    false

@addUniqueIndex = (uniqueIndex) ->
  if uniqueIndex instanceof UniqueIndex
    _uniqueIndices.addItem(uniqueIndex)
  else
    false

@allowPkInsert = () -> _attributes.allowPkInsert

@getBaseClass = () -> _attributes.baseClass

@getBasePeer = () -> _attributes.basePeer

@getDescription = () -> _attributes.description

@getIdMethod = () -> _attributes.idMethod

@getColumns = () -> _columns.getItems()

@getForeignKeys = () -> _foreignKeys.getItems()

@getIndices = () -> _indices.getItems()

@getUniqueIndices = () -> _uniqueIndices.getItems()

@getName = () -> _attributes.name

@getNamespace = () -> _attributes.namespace

@getPackage = () -> _attributes.package

@getPhpName = () -> _attributes.phpName

@getPhpNamingMethod = () -> _attributes.phpNamingMethod

@getSchema = () -> _attributes.schema

@getTreeMode = () -> _attributes.treeMode

@isAbstract = () -> _attributes.abstract

@isCrossRef = () -> _attributes.isCrossRef

@isHeavyIndexing = () -> _attributes.heavyIndexing

@isReadOnly = () -> _attributes.readOnly
```

```
@isSkipSql = () -> _attributes.skipSql

@reloadOnInsert = () -> _attributes.reloadOnInsert

@reloadOnUpdate = () -> _attributes.reloadOnUpdate

@removeColumn = (column) ->
  if column instanceof Column
    _columns.removeItem(column)
  else false

@removeForeignKey = (foreignKey) ->
  if foreignKey instanceof ForeignKey
    _foreignKeys.removeItem(foreignKey)
  else
    false

@removeIndex = (index) ->
  if index instanceof Index
    _indices.removeItem(index)
  else
    false

@removeUniqueIndex = (uniqueIndex) ->
  if uniqueIndex instanceof UniqueIndex
    _uniqueIndices.removeItem(uniqueIndex)
  else
    false

@setAbstract = (bool = true) ->if bool
  _attributes.abstract = if bool then true else false

@setAllowPkInsert = (bool = true) ->
  _attributes.allowPkInsert = if bool then true else false

@setBaseClass = (baseClass) ->
  baseClass = baseClass.trim() if typeof baseClass is "
     string"
  _attributes.baseClass = if baseClass then baseClass else
     Table.INHERIT

@setBaseClass = (basePeer) ->
  basePeer = basePeer.trim() if typeof basePeer is "string"
  _attributes.basePeer = if basePeer then basePeer else
     Table.INHERIT

@setDescription = (description) ->
  _attributes.description = if description then description
```

```
      else null

  @setHeavyIndexing = (bool = true) ->
    if bool is Table.INHERIT then _attributes.heavyIndexing =
       Table.INHERIT
    _attributes.heavyIndexing = if bool then true else false

  @setIdMethod = (method = Table.INHERIT) ->
    if method in Table.IdMethod.toArray() then _attributes.
       method =  method else Table.INHERIT

  @setIsCrossRef = (bool = true) ->
    _attributes.isCrossRef = if bool then true else false

  @setName = (name) ->
    name = name.trim() if typeof name is "string"
    _attributes.name = if name then name else throw "Table
       must have a name"

  @setNamespace = (namespace) ->
    namespace = namespace.trim if typeof namespace is "string"
    _attributes.namespace = if namespace then namespace else
       Table.INHERIT

  @setPackage = (thepackage) ->
    thepackage = thepackage.trim if typeof thepackage is "
       string"
    _attributes.package = if thepackage then thepackage else
       Table.INHERIT

  @setPhpName = (name) ->
    name = name.trim() if typeof name is "string"
    _attributes.phpName = if name then name else null

  @setPhpNamingMethod = (method = Table.PhpNamingMethod.
     UNDERSCORE) ->
    if method in Table.PhpNamingMethod.toArray() then _method
       = method else Table.INHERIT

  @setReadOnly = (bool = true) ->
    _attributes.readOnly = if bool then true else false

  @setReloadOnInsert = (bool = true) ->
    _attributes.realoadOnInsert = if bool then true else false

  @setReloadOnUpdate = (bool = true) ->
    _attributes.realoadOnUpdate = if bool then true else false
```

```
@setSchema = (schema) ->
  schema = schema.trim if typeof schema is "string"
  _attributes.schema = if schema then schema else Table.
    INHERIT

@setSkipSql = (bool = true) ->
  _attributes.skipSql = if bool then true else false

@setTreeMode = (treeMode) ->
  _attributes.treeMode = if treeMode in Table.TreeMode.
    toArray() then treeMode else null

@setName name
```

2.2 CoffeeScript Inheritance — jQuery extension

This code example shows how namespaces were implemented in the UI library. It demonstrates the work-around for jQuery's lack of OO design which can be seen in the `decorate()` function which adds its overriding functions to the `this` ($with is effectively `this` in the decorate function). These functions are the jQuery functions such as html(), text() and prepend().

It has more prototypical functions than the previous example as it requires no privileged functions.

This example has been subjected to line wrapping. Whitespace in CoffeeScript is significant and, unfortunately, the wrapping here would not constitute well-formed CoffeeScript code.

```
window.Bootstrap or=
  Components:
    Alert: class Alert extends jQuery
      constructor: (text) ->
        Alert.decorate $("<div>"), this
        $button = $ "<button>", type: "button", "data-dismiss":
           "alert"
        $button.addClass "close"
        $button.text "x"
        this.append $button

      alertBlock: (alertBlock = true) ->
        this.addClass "alert"
        if alertBlock
          $this.addClass "alert-block"
        else
          $this.removeClass "alert-block"

      setContext: (context = "warning", message) ->
        contexts = ["error", "info", "success", "warning"]
        this.removeClass "alert-#{contexts.join " alert-"}"
        if context not in contexts then context = "warning"
        this.addClass "alert alert-#{context}"
        this.html message if message?
        this

      danger: (message) -> this.setContext "error", message

      @decorate: ($item, $with = new Alert()) ->
        if $item not instanceof Object then $item = $ $item
        $.extend $with, $item
        $with.error = (message) -> Alert.prototype.error.call
           this, message
        $with.html = (html) -> Alert.prototype.html.call this,
           html
        $with.prepend = () -> Alert.prototype.prepend.apply this
           , arguments
```

```
    $with.remove = () -> Alert.prototype.remove.call this
    $with.text = (text) -> Alert.prototype.text.call this,
      text
    $with.addClass "fade in"
    Alert.prototype.setContext.call $with, "warning"

  error: (message) -> this.danger message

  html: (html) ->
    $button = this.children(".close").first()
    $button.detach()
    if typeof html is "function" then html = html.call this,
      0, this.get(0)
    this.empty().append $button, $ $.parseHTML html

  info: (message) -> this.setContext "info", message

  prepend: () ->
    item = []
    if arguments.length > 1
      item.push arguments[arg] for arg of arguments
    else if typeof arguments[0] is "function" then item =
      aguments[0].call this, 0, this.get(0)
    else item = arguments[0]
    $button = this.children(".close").first()
    $siblings = this.detach(".close:first").contents()
    this.empty().append $button, item, $siblings

  remove: () ->
    if typeof this.alert is "function"
      this.alert("close") # for bootstrap-alert.js
    else
      this.remove()

  success: (message) -> this.setContext "success", message

  text: (text) ->
    $button = this.children(".close").first()
    $button.detach()
    if typeof text is "function" then text = text.call this,
      0, this.text()
    this.empty().append $button, document.createTextNode
      text

  warning: (message) -> this.setContext "warning", message
```

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.