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 15, 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 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 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 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

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 Standards

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

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.

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. This has been included in the project against the principles of agile development as it comes with the Initializr package.

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.

17 of 21

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];
```

18 of 21

```
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)
{
```

19 of 21

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

20 of 21

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.

21 of 21