

Object Resource Manager and its Application to a Sea Turtle Virtual Environment

Joel Sutherland

Honors Thesis
Department of Computer Science
UNC-Chapel Hill
May 2007

The Turtle Project

Background

Shortly after birth, Loggerhead turtle hatchlings are able to cover over 9,000 miles in a large migratory path that covers the Atlantic Ocean. The Lohmann Lab in the UNC department of biology is exploring the mechanisms these turtles have that enable them to navigate the vast and featureless ocean. Current research done by Dr. Ken Lohmann suggests that the Loggerhead turtles are able to sense some properties of the Earth's magnetic field that allow the turtles to orient themselves on the globe. (<http://www.unc.edu/depts/oceanweb/turtles/>)

Dr. Lohmann is currently studying this magnetic sense. A special tank has been constructed that holds a turtle; it is surrounded by wires that are driven by custom hardware, producing a magnetic field that simulates that of the Earth. A sensor records which direction the turtle is swimming. In past successful experiments, this setup has allowed researchers to place turtles in simulated geographic locations along their migratory route and observe the direction that they swim. The goal of this project is to allow researchers to track these turtles as they perform a full migration of the Atlantic Ocean and visualize the results on a map.

While the lab has simulated small pieces of a migration, they now want to run experiments that show a turtle doing a full migration. Software has been written that controls the magnetic field around the tank and observes the direction the turtle is swimming. As the turtle continues to swim, the program changes the magnetic field to reflect the turtle's current location on a simulated Earth. Data from this software is then written to a log file.

Purpose

The primary purpose of this project is to read the data generated by the current software about the turtle's location, and visualize it on a map. This will allow researchers to get a better understanding of an experiment's progress as it happens, and also provide a visual representation of an experiment for use in publications. Secondary goals of the project are to provide an interface to manage various experiments and trials, monitor a turtle's progress and alert an experimenter when events he is interested in happen and provide a facility to export data.

Experiment Setup and Management

Structural Organization

Each experiment that is created will hold three types of information. First will be general experiment settings. This will include information such as an experiment name and information about the experiment such as turtle speed and what type of experiment is taking place. The two types of experiments that can take place are either on a true simulation of the earth, or on an arbitrary "magnetic" map, where the coordinates represent some property of the magnetic field. The second type of information will identify the type of experiment that takes place such as what events are of interest and actions to perform when the events happen. The final type of information will be the actual data that is collected in a series of trials. The first two types of information can be saved as a template so that additional experiments using the same setup will be possible without data re-entry.

Event Management

The actions that take place during an experiment will be planned and controlled using an event management system. This system is composed of a series of event triggers that cause an action to take place. An event trigger might be that a turtle enters or exits a geographical region. Events will be stored as PHP scripts and additional events can be written and added to the system by saving them in the events directory. When an event occurs, it can cause a single action, or a series of actions, to happen. These actions can include things such as: dropping a food pellet, flashing a light, resetting the location of the turtle, or starting a new trial. The scope of this project is limited such that the actions that require an interface to hardware will instead cause a prompt to appear on the screen for an experimenter to perform the action.

Data Storage and Format

Data associated with an experiment will be stored in a unique experiment directory. This directory will hold a log file for each trial in the format that the existing software creates at a resolution specified in the experiment setup. Additionally, data that is stored about the experiment setup and individual trial runs will be stored in an embeddable SQLite database. (<http://www.sqlite.org/> or <http://us3.php.net/sqlite>)

Reports

Data that has been collected from experiments will be available in a raw format or through a report filter. A report filter is a script that will analyze and return calculated data. Any number of scripts can be uploaded and used as a report filter. Additionally, an experiment overview sheet will be available for printout that will show a map of the results along with a summary of the experiment setup.

Interface

The user interface to this part of the project is entirely web-based. Experiment setup data is viewable and editable through HTML forms. Maps are shown using the Google™ Maps API on a simple web page.

Issues and Decisions

Interfacing with Current Software

Caretta, the software that interfaces with the special tank hardware, is written primarily in C# and C++ and uses some C libraries. It performs hardware calibration, monitors the turtle and its location on a simulated Earth, and produces a log file of the turtle's journey in real time. This software was written primarily in a semester by Patrick Reynolds as an undergraduate project and is maintained and has been extended by Mason Drew. Interfacing with the hardware was a challenge to get perfect and was described as being very complicated. Since Caretta fulfilled its designed purpose, it seemed unwise to try to directly extend it to add visualization capabilities. Instead it was decided to use the log file as the primary interface between Caretta and this project. A simple message passing scheme is used when the project needs to ask Caretta to perform an action.

Choosing a Platform

Once it was determined that there would be very little direct interaction between Caretta and the new software, the platform possibilities were wide open. The author's personal experience directed him towards Java and PHP. His background in Java involved using AWT to create an interface for restaurant management software. In PHP, he had helped develop a framework, OpenScaffold, and worked on a number of larger applications. Ultimately the web route was chosen.

There were two motivations for making this project an online application. The first was development simplicity and the second was public access. It was decided that it would be simpler to develop this project as an online application as it would be visually complicated and the web, HTML, and CSS would allow for the simple creation of a visually complex application. Additionally, much of the heavy lifting could be offloaded to Google Maps. (<http://maps.google.com/> and <http://www.google.com/apis/maps/>) Public access was also a consideration as it may be beneficial to allow the public to view data and progress on experiments for the purposes of grant fulfillment.

Using OpenScaffold

Knowing that the web would be used and that the application would be written in PHP, it made the most sense that it be written on OpenScaffold.

(<http://www.openscaffold.com/>) OpenScaffold is a PHP framework designed for rapid application development. Like other frameworks, it handles many repetitive and unnecessary tasks that a developer must deal with when creating an application including: data persistence, URL routing, session handling, and templated views. The framework was created by Kris Jordan and the author primarily outside the classroom with the exception of the data persistence layer, Core, which was developed as a part of an independent study. The Core User Guide is included as Appendix 2.

The primary reason that OpenScaffold was used for this project above other frameworks was the author's familiarity with it. There are alternative frameworks in PHP, but most are unwieldy and designed for much more complicated projects. Web frameworks written in languages other than PHP, such as Ruby on Rails and Django (Python), were not considered as the time spent learning the language and framework would limit the time spent building the actual application.

Displaying Maps

A critical part of the turtle project is the visualization of the data. The visualization has several requirements: it needs to be accurate, to update in real-time, to show geographic features to put the turtle's path in perspective, to look nice when printed for reports. and finally, to be usable offline when researchers are using a laptop in the field.

These requirements presented several challenges. A turtle's path needs to be observed on widely varying scales and presented against any part of the globe. At times it will be traveling along a portion of the Florida coast; at others it would be migrating the Atlantic. The Earth's magnetic field is particularly interesting in the Indian Ocean (where the field is nearly a grid) so it needed to be shown there as well. Displaying this much geography in the required detail necessitated the use of very large texture files. If it were not for the offline access requirement, Google Maps was the obvious choice. It provided a solid API and did much of the heavy lifting required for the texture processing. In order to meet the offline requirement, emulation of the Google interface was considered. It was determined, however, that the required texture work would be prohibitively slow. For this reason a compromise was reached. For the field, a simple version will be developed. Once back in the lab, a nice view and interface to the data would be provided through Google Maps. Google Maps would also provide a much more interesting public web display of the experiment, including the map zooming and re-centering that people are used to.

Handling Events and Actions

Another aspect of the project's design that needed to be considered was how the software should observe and respond to a turtle's progress. Preliminary conversations about potential experiments that would be run using the software led to a couple of possible scenarios. In one, the system would need to recognize when a turtle had entered a certain area and reward it with food; in another, the system would need to know when a turtle had been traveling in a certain direction for a given amount of time. It was recognized that not all situations could be anticipated. For this reason we decided that a super user would be able to define new events that could be recognized and new actions that could take place by writing PHP scripts that fit a defined format. These would simply be placed in a folder and then be accessible to regular users as options when defining experiments.

Working with Google Maps

How Maps are Displayed

The interface that Google provides to its map application solves many performance issues that would present themselves if a developer needed to run all aspects of the map rendering on a local machine. The process that it uses to present its display is simple. Through JavaScript, simple calls are made to the Google server that request certain views to be rendered. In the case of a simple map display, Google returns a simple, pre-rendered, compressed image of the area requested at the zoom level requested. Google has a strong advantage in creating this image: due to the traffic it receives, it is likely cached. Doing this procedure locally would mean dealing with texture files that could reach over several gigabytes in size once decompressed. As the view is zoomed or relocated, Google is able to seamlessly pass new image files.

Google handles rendering overlays in a slightly different manner. In the typical case, information about shapes and lines are passed to Google servers using JavaScript. Google then renders a png file (to allow for transparency) of the appropriate size for the current zoom level. At this point, SVG, a browser graphics language, (<http://www.w3.org/Graphics/SVG/>) is used by JavaScript to overlay the transparent image and relocate it as the user drags the map around. For very large lines or polygons of many points, this takes a large burden off the local machine. It does however reduce response time from the user's perspective. Any change to the line or the zoom level requires communication with Google and the transfer of a new image file.

Refresh Performance

Ultimately, two major types of display refreshes occur in the mapping program. Initially, the entire path of the turtle is loaded from the log file that is to be displayed. This requires accessing the database and selecting a row for each log entry and then displaying that data as XML. Currently this process is done

through OpenScaffold with the XML rendered by the Smarty Templating System. (<http://smarty.php.net/>) If performance with this part of the operation ever becomes an issue, it may be wise to store the XML file and only append new data to it, as opposed to the current strategy of rendering it on each request.

Once the initial display has been loaded, the id of the last drawn data point is stored in the HTML view. At each refresh interval, this id is passed into the application and an XML file is rendered and returned that contains all new data points. This is drawn onto the Google map as a new polyline, rather than appending to the existing polyline. Since this only deals with a limited amount of data, the refresh time is quick and no additional performance enhancements in the future should be necessary.

Polylines

The Google Maps API provides a number of convenient features for this project. The existing software that interfaces with the data collection hardware produces a log file that contains a series of coordinates. The Google Maps API provides three methods for plotting a polyline that consists of a series of coordinates. Ultimately, the XML method of rendering the lines was selected, but all three methods are discussed below.

Basic Polylines: Simple Polylines can be created in JavaScript when building a Google Maps object. A polyline object consists of an ordered list of point objects. Each point object consists simply of a latitude and longitude value. The advantage to this method is that it is extremely simple to use. The disadvantage is in performance. A turtle's path may consist of hundreds or even thousands of points. Creating a JavaScript object for each is inefficient and may cause browser instability. The problem is exacerbated in Internet Explorer (IE). Typically, the polyline data is sent to Google, rendered into a line and sent back as a part of the map texture. In IE however, the line is rendered client-side using SVG. This creates exceptional overhead for the client and causes performance issues even with as few as 200 points.

Encoded Polylines: The newest and most promising method of displaying a complex polyline is through an encoding that Google describes in its API. It essentially allows a string to be passed into the Google Maps object that consists of an ASCII encoding of the points in the line. The encoding also can include extremely useful information about which points to display at each zoom level. The only downside to this method seems to be temporary. This is newly released into the API and errors have already been discovered in its documentation. This has made proper and efficient encoding impossible. In the future, however, it is likely that this will be the preferred method of creating polylines.

XML Polyline: Google provides an alternative to the basic polyline in its API that involves an asynchronous call to a remote XML document that contains each of the points. While its documentation does not advertise its implementation method, investigation reveals that this process simply creates polylines through the basic process described above, offering no performance improvement. In fact, the XML overhead may cause a decrease in performance. This method can be useful to update a line without redrawing all the points.

This method was selected since it allowed for the easiest and most efficient redraw of points. Though it uses the slower basic polyline method of drawing the points, it is easier to append more data to an existing line by simply creating a new line. Since the display will be refreshed about once every 15 seconds, each new appended line will contain only 15 points, making any performance hit negligible.

Handling Events and Actions

System Requirements

One of the strongest values the system provides is the ability to recognize behavior of the turtle and respond by triggering an action. For instance, it might be useful to notice that a turtle has entered a certain region and reward him. The types of events that will need to be recognized and actions that will then take place may change as new experiments are designed or old ones are changed. For this reason, it makes sense to build the event recognition and action response system to allow for easy extendability.

Relationship between Experiments, Trials, Events and Actions

Experiments *have many* Events and *have many* Trials
Events *have many* Actions
Events and Actions *have many* Properties

Each experiment can have any number of events associated with it. During a trial, the experiment's events will be displayed using their display property, and their conditions will be monitored to check if an event has taken place. Once an event occurs, its actions are triggered.

To make sure that a user has enough freedom when designing their events and actions it is important that we allow any number of properties to be gathered and used. In the example of an event that determines whether a turtle has entered a radius, the event would need at least three properties: one for latitude, one for longitude and one for the circle radius. We can imagine that more complex events or actions could be designed that needed more properties that might not be structured so simply. For example, an experimenter may only want to trigger the event if the turtle stays within the circle for a certain amount of time. For this reason, we will store all properties of an action as a serialized object in a single field of the event or action table. This will allow great flexibility without hurting

performance as only a few actions or events will need to be monitored at any given time.

Designing Events

Designing an event will simply require the creation of a PHP file in the events directory. The system will look for and notice all valid event scripts that are added to this directory and then allow them to be used in the event creation process.

Since event recognition will require the monitoring of data, the following variables will exist for an event creator to use:

- **\$events:** array of events associated with this experiment
- **\$current_location:** the log entry of the turtle's current position
- **\$recent_history:** array of the last 100 log entries.
- **\$sampled_history:** array of 100 log entries that evenly cover the turtle's path up until this point. The entries are selected, not computed.

Each log file has the following properties available:

- **Latitude**
- **Longitude**
- **date_created**
- **trial_id**

Any number of calculations can be done on these variables. For convenience, the following collection of functions is available:

- **vectors(\$log_entries):** takes an array of log entries and returns an array of vectors
- **in_location(\$entry, \$latitude, \$longitude, \$radius):** takes a log entry and a geographic point and determines whether the point is within the radius from that point
- **draw_circle(\$lat, \$lng, \$rad):** draws a circle on the map based on the given center point and radius in kilometers
- **draw_box(\$tlLat, \$tlLng, \$brLat, \$brLng):** draws a box on the map based on the given top left and bottom right corner coordinates
- **draw_poly(\$points[], \$color...):** draws a closed polygon on the map based on the array of passed points and the fill color
- **draw_line(\$points[]):** draws a polyline along the array of given points

Creating Event Classes

Creating an event involves writing a class that fits a certain structure and placing it in the events directory. The system will then recognize that an event has been added and will allow experimenters to select the event when they set up an experiment.

There are three main things that need to be defined when creating an event: the properties of the event, displaying the event, and definition of its occurrence.

First, the “__construct()” function is used to define the properties that the event will need. There can be any number of properties, but every event must have the “draw” property. This allows an experimenter to decide whether or not the event should be drawn on the map. This is useful so that many events can be set up on an experiment without cluttering the display.

The second part of the event class is the “display()” function. This function consists of JavaScript that will be inserted into the map display to show the event. It is useful to look at the Google Maps API to see the different functions that are already available to draw things on the map.

Finally, the most important piece of the event setup is the “occurred()” function. This function needs to return either true or false, depending on whether the event has occurred. To determine if an event has occurred, use the variables described above and add logic.

Here is an example of an event.

```
class EnterCircle extends Event{
    public $properties;

    //This defines the properties required with the event
    public function __construct(){
        $this->properties = array('draw', 'latitude', 'longitude', 'radius'...);
    }

    //This function is called when the map is initially drawn
    public function display(){
        return 'alert("Hello World")';
    }

    //This returns true if the even has occurred.

    public function occurred(){
        return in_location($current_location, $this->radius);
    }
}
```

Designing Actions

Just as each experiment can have any number of events attached to it, each event can have any number of actions attached. When the event occurs, each

action that is attached will execute. Like events, actions can be scripted using a simple class format. In many ways, they are easier than events since no logic needs to be written to determine whether or not to run them. Actions also are going to be doing a much wider variety of things, like interfacing with hardware that runs a light or that drops food pellets. For this reason there are fewer variables and functions that need to be considered when creating an action.

Creating Action Classes

A file that is created containing an action class will be recognized by the system as soon as it is added to the actions folder in the app directory. The format required for an action class is simple, there are just two required functions.

The first function, "`__construct()`" works just like it does for events. It defines the properties that the action will need and that the user will be able to edit when adding an action to an event. Unlike the event classes, there is no required property 'draw' since many actions may not need to be reflected visually on the map.

The second function is "`act()`", which is called with the event the action is attached to occurs. This function may need to access an external api or data that the system has collected. If it needs to get at data, model classes from the system will have to be imported. For more information on how to do this as well as examples, look at the controller classes as well as the OpenScaffold website (<http://www.openscaffold.com/>) and the Core documentation. Finally, whatever string the '`act()`' function returns is sent to Google Maps and is displayed in an alert.

Example of an action.

```
class Alert extends ActionHandler{
    public $properties;

    //This defines the properties available to a user when creating an action
    public function __construct(){
        $this->properties = array('message');
    }

    //This function is called when the action needs to act
    public function act(){
        $script = $this->message;
        return $script;
    }
}
```

Resources

Google Maps API: <http://www.google.com/apis/maps/>

Lohmann Labs: <http://www.unc.edu/depts/oceanweb/turtles/>

OpenScaffold: <http://www.openscaffold.com/>

Core: <http://www.openscaffold.com/>

PHP: <http://www.php.net/>

PHP5: <http://www.php.net/manual/en/>

Smarty: <http://smarty.php.net/>

SQLite: <http://www.sqlite.org/>

PHP5 & SQLite: <http://us3.php.net/sqlite>

PDO SQLite: <http://us3.php.net/manual/en/ref.pdo-sqlite.php>

SVG: <http://www.w3.org/Graphics/SVG/>

Appendix 1: Installation

Installing the visualization application requires that a computer be running a web sever with PHP installed. The following instructions explain how to set up Apache and PHP as well as install dependencies on a Windows XP or greater machine.

1. Install Apache HTTP Server

1. Visit <http://www.apache.org>
2. Download the latest 2.0 version of Apache (not 2.2!)
3. In Windows XP Double Click to install.

In Vista, permissions must be set to allow Apache to run as a service. To do that: Go to Start | All Programs | Accessories, then right-click on "Command Prompt", and select "Run as administrator". This will open a command prompt window with admin rights. Navigate to the directory where you saved the apache*.msi install file, and type "msiexec /i apache*.msi" (where apache*.msi is the actual name of the file - hint: type apache and then hit tab, it should fill in the file name automatically).

1. Choose a root directory for the whole project.
2. Install Apache to a Servers folder within that directory.
3. When installation is complete, navigate to Root Folder/Servers/Apache/conf, and replace httpd.conf with the attached file.
4. Update all paths within httpd.conf to reflect your directory structure.

2. Install PHP

1. Visit <http://museum.php.net/php5> (You need to go to the archives to get 5.1.)
2. Download the latest 5.1.x release as a zip file. (As of this writing, php 5.2 has a bad PEAR install - this will likely be corrected)
3. Extract the PHP folder in the zip file to Root Folder/Servers
4. Navigate to Root Folder/Servers/PHP and add the php.ini file (<http://www.openscaffold.com/install/>)
5. Update all the paths in the php.ini file

3. Install PEAR

1. Go to Start | Programs | Accessories | Command Prompt
2. Navigate to the PHP folder
3. Run go-pear.bat accepting the default for all questions

4. Install Application

1. Extract the openscaffold folder to Root Folder / SVN /
2. Create PHPSessions and PHPFiles in Folder Root/Servers
3. Update the database path in the config.php file

Appendix 2: Core User Guide

Purpose of this Guide

This guide is intended to introduce new users of Core to its features. Core users are expected to be familiar with php, object-oriented programming and the Model-View-Controller architecture to be able to use this document. Additionally they should have already set up the Core environment.

Introduction to Core

Core is intended to eliminate the persistence headaches caused by using relational databases in Object Oriented programs by raising the level of the database interface. It allows the user to create a series of model classes that extend from Core that provide access to a database. It is written in PHP5 and uses PDO drivers to interact with the databases.

Core saves time and eliminates headaches by changing the way that the model layer is built. Instead of defining a number of operations on the database, instead you specify the relationships between objects. This allows Core's default behavior to be smart, giving you the information you want when you retrieve objects and persisting the correct objects when you save objects.

The primary design goal of Core was to save developer time. To do this, Core relies on conventions so that things only need to be specified once. These conventions are listed at the end of this document for convenience.

Getting Started

To get starting creating a project that uses Core, you must first see the installation document. The next step is creating the database and tables you wish to have access to in your project. See the end of this document for naming conventions when you create the database and tables.

The next step is to begin defining relationships by creating model classes. Once nice feature of Core is that it is not necessary to specify anything about standard fields in the database. When Core does a retrieve or persist it looks at the database to know which fields to make available. The purpose of the model classes is to define the relationships between individual tables. The following examples show how this works.

One-to-One Relationships

Model Class

This type of relationship involves rows of two tables that have a one-to-one correspondence. An example would be the relationship between an employee and their company car. For example, “Bill Gates”, an employee in the employees table would be associated with just one row of the company_car table, Mercury Sable id 49. The code to accomplish this association in Core would look as follows:

```
class Employee extends Core{
    public function __construct(){
        $this->BelongsTo('company_car');
    }
}

class CompanyCar extends Core{
    public function __construct(){
        $this->HasOne('employee');
    }
}
```

It is worth noting the difference between “HasOne” and “BelongsTo” in this example. Just as in Rails’ Active Record, the notion of “BelongsTo” implies that a foreign key exists in that table. In this example it means that in the employees table, there is one column called “company_car_id”.

Using the Model Class

After creating the model class it will be useful to use it in your program. The following example shows how one would find an employee and access information about their company car.

```
$searchEmployee = new Employee();
$searchEmployee->name = "Bill Gates";
$results = Employee::find($searchEmployee);
$billG = $results[0]; //Takes first result
print $billG->name; //Prints "Bill Gates"
print $billG->company_car->type //Prints "Mercury Sable"
print $billG->company_car->employee->name //Prints "Bill Gates"
```

Note that we can search on any properties by passing an object of the Model type to the static method find(). This returns an array of the results. Also notice the second to last line of code. The \$billG object has built in access to all of the properties of its relationships.

One-to-Many Relationships

Model Class

A one-to-many relationship involves a row of one table corresponding to many rows of another table. An example would be the relationship between an employee and their payroll checks. For example, “Bill Gates”, an employee in the employees table would be associated with many (MANY) payroll checks from the checks table. The code would be as follows:

```
class Employee extends Core{
    public function __construct(){
        $this->BelongsTo('company_car');
        $this->HasMany('checks');
    }
}

class Checks extends Core{
    public function __construct(){
        $this->BelongsTo('employee');
    }
}
```

Using the Model Class

A one-to-many relationship works much the same in practice as a one-to-one relationship.

```
$searchEmployee = new Employee();
$searchEmployee->name = "Bill Gates";
$results = Employee::find($searchEmployee);
$billG = $results[0]; //Takes first result
$sum = 0;
foreach($billG->checks as $check){
    $sum += $check->amount;
    print $check->amount; //Prints each check amount
}
print $sum; //Prints total earnings
$newCheck = new Check();
$newCheck->amount = 1000000.00
$billG->addCheck($newCheck) //Persists a new check
```

So just as in the one-to-one example, we can access a one-to-many relationship directly. The difference is that it returns an array. We are also able to directly add Checks to Bill’s payroll information. The “addCheck()” method determined through reflection and is available for all one-to-many relationships.

Many-to-Many Relationships

Model Class

A many-to-many relationship is much different in the way that is implemented than a one-to-one or one-to-many relationship. They all talk about the relationship between two tables, but it requires three tables to implement a many-to-many relationship. The third table is called an index table and each row stores two foreign keys, one from each table. In order to provide the most flexibility, a model class must be created for this index table. This table could be implied, but by making it explicit we are able to add properties to the index table. An example of a simple many-to-many relationship would be the relationship between an employee and their current projects. For example, “Bill Gates” an employee from the employees table, works on many projects that have many other employees working on them. The code to create these models is as follows:

```
class Employee extends Core{
    public function __construct(){
        $this->BelongsTo('company__car');
        $this->HasMany('checks');
        $this->HasMany('projects', new Through('assignments'));
    }
}

class Assignment extends Core{
    public function __construct(){
        $this->BelongsTo('employee');
        $this->BelongsTo('project');
    }
}

class Project extends Core{
    public function __construct(){
        $this->HasMany('employees', new Through('assignments'));
    }
}
```

There are a couple of things to note in this code example. First, we can see the addition of a new parameter to the HasMany constructor. The addition of new Through, tells Core which class describes the index table. Notice that this was added to the HasMany of both Employee and Project.

Another item worth noting is the BelongsTo relationships in the Assignment class. Just as in one-to-one and one-to-many relationships, this tells Core that there are foreign key columns in the Assignment table that point to the appropriate rows in the other two tables. This of course is the purpose of an index table.

Also note that since we are explicitly declaring a model class for the index table, we are able to add attributes to the assignment relationship between employees and projects. For example, we could add a HasMany relationship between the assignments index table and a tasks table to attach tasks to an assignment.

Using the Model Class

Many-to-many gives us much of the same functionality as one-to-many, but in both directions.

```
$searchEmployee = new Employee();
$searchEmployee->name = "Bill Gates";
$results = Employee::find($searchEmployee);
$billG = $results[0]; //Takes first result
foreach($billG->projects as $project){
    print $project->title . ":"; //Prints each project title
    foreach($project->employees as $employee){
        print $employee->name; //Prints each employee
        of the
    } project
}
```

This code goes through all of the projects Bill is assigned to and prints the project title followed by the name of every employee working on the project(including Bill). Notice that accessing the many-to-many relationships is not different than accessing the one-to-many relationships.

Explicit Relationship Naming Example

Model Classes

Often it is necessary to work in a situation where it is not possible to follow all naming conventions. In this situation Core allows users to break conventions by being explicit about naming when declaring relationships. If in the one-to-many checks example above, for example, the checks table was instead named payroll_checks, it would be an inconvenient and lengthy name for an attribute of person. Instead we can explicitly name the relationship 'checks' and use the Table parameter to specify the target table's name. The following example shows how.

```
class Employee extends Core{
    public function __construct(){
        $this->HasMany('checks', new Table('payroll_checks'));
    }
}

class PayrollChecks extends Core{
    public function __construct(){
        $this->BelongsTo('employee');
    }
}
```

Now that this is done, an employee object has access to rows in the payroll_checks table associated with him through the checks attribute.

Targeting a Table Twice Example

Model Classes

Sometimes it is necessary to have more than one relationship between two objects. In this case, it is not possible to stick to naming conventions when it comes to foreign keys. For example, imagine that each computer at a company has both a user and a maintenance employee assigned to it. This relationship would be set up as follows:

```
class Employee extends Core{
    public function __construct(){
        $this->HasOne('computer', new Key('user_id'));
        $this->HasMany('supportedComps', new Table('computers'),
            new Key('supporter_id'));
    }
}

class Computer extends Core{
    public function __construct(){
        $this->BelongsTo('user', new Table('employee'), new Key('user_id'));
        $this->BelongsTo('supporter', new Table('employee'),
            new Key('supporter_id'));
    }
}
```

This example makes it clear why explicitly declaring keys is necessary when there is more than one relationship between objects. Note that it is important to declare this in both directions so that lookups from either object operate correctly.

Family Example

The following example uses several different types of relationships to create the Model classes for a family. This example will also demonstrate the use of some of the additional naming features. Because this example is particularly complex, it will be illustrative to show the way that the tables are set up in the database.

Tables

The following three tables with the shown properties are needed for this example.

people	progeny	marriages
id name	id parent_id child_id	id self_id spouse_id date_start date_end

Model Classes

The relationships we are going to set up will describe a family. For the purposes of creating a complex example our notion of a family is progressive in nature. Sex is not defined in the marriage relationship. Additionally a marriage has a start and end date, allowing for divorce, remarriage and multiple simultaneous marriages.

```
class Person extends Core{
  public function __construct(){
    $this->HasMany('children', new Through('progeny'),
                                     new Key('parent_id'));
    $this->HasMany('parents', new Through('progeny'),
                                     new Key('child'));
    $this->HasMany('spouses', new Through('marriages'),
                                     new Key('self'));
    $this->HasMany('marriages');
  }
}

class Progeny extends Core{
  public function __construct(){
    $this->BelongsTo('parent', new Table('People'));
    $this->BelongsTo('child', new Table('people'));
  }
}

class Marriages extends Core{
  var $date_start; //Note this would be implied
  var $date_end;   //but are included for clarity
  public function __construct(){
    $this->BelongsTo('self', new Table('people');
    $this->BelongsTo('spouse', new Table('people'));
  }
}
```

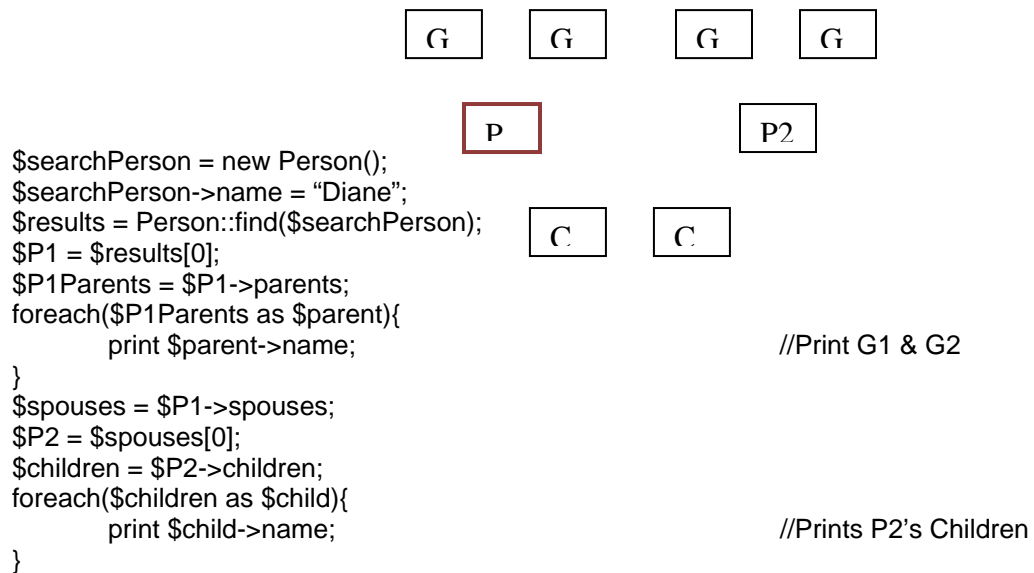
This example is different from the earlier example because the many-to-many relationships only have two tables, an index table, and the Person table that relates to itself. Relating a table to itself requires using a new relationship modifier called "Identifier". This modifier tells core which column to identify with

in the index table. For example, if we were trying to find a person's parents, we would need to look in the progeny table and know that we are looking from a Child's perspective.

The other modifier introduced in this example is the "Table" modifier. This tells core which table the relationship lives in. Normally, Core would know to look for an "Employee" in the "employees" table. In this example, however Progeny has to find "Parent", but it can not find it in the "parents" table. We can tell it to look in the "people" table by using the "Table" modifier.

Using the Model Classes

Using the classes we created above we will traverse a family tree, starting at just one person, P1 whose name is Diane. For reference the family tree looks as follows:



Using Model Classes – Magic Methods

You may have noticed a couple of interesting methods being called in some of the above examples. One feature of Core is that it creates some "Magic Methods" to make some operations on one-to-many and many-to-many relationships easier. Specifically, core creates "add" and "delete" methods that allow you to add and delete objects from collections.

So, for example, say we have a one-to-many relationship that between employees and checks, just as we do above. We could do the following things:

```

$searchEmployee = new Employee();
$searchEmployee->name = "Bill Gates";
$results = Employee::find($searchPerson);
$billG = $results[0];
// Now we can list Bill's checks
foreach($billG->checks as $check){
    print $check->id . " " . $check->amount;
}
$id = 4;
// Using a magic method we can delete check with id 4
$billG->deleteFromChecks($id);
// Also using a magic method we can add a check
$check = new Check();
$check->amount = 500;
$billG->addToChecks($check);

```

These two operations give a handy way to deal with the many items in a relationship. The delete magic method works as expected. The add method does as well, handling several things for you. It makes sure to record all the foreign keys necessary to make the addition possible and assign a good id to the new check.

Syntax Description

The following is a brief dictionary-style description of the different syntax and terms used in core.

Core

This is the base model class that individual model classes extend from. It provides all of the built in functionality and relationship mapping. Usage:

```
class Person extends Core{ ... }
```

__construct()

This is a php5 constructor. It is here that all relationship declaration methods must be called. Usage:

```

class Person extends Core{
    public function __construct(){ ... }
}

```

Relationship Declaration Method

These are the methods that are to be called in a Core model's constructor. They declare relationships to other objects. The first attribute to a relationship declaration method is the name of the relationship. The following methods are available:

HasOne() – Used to specify a relationship to a single row in another table. Implies that the foreign key exists in the other table. It is convention for the relationship name to be the singular form of the foreign table name.

HasMany() – Used to specify a relationship to multiple rows in another table. Also implies that the foreign key is stored in the other table. It is convention for the relationship name to be the foreign table name.

BelongsTo() – Used to specify a relationship to a single row in another table. Indicates that the id of that foreign row is stored in the local table. It is convention for the relationship name to be the singular form of the foreign table name.

Usage:

```
class Person extends Core{
    public function __construct(){
        $this->HasOne('address');
        $this->HasMany('checks');
        $this->BelongsTo('department');
    }
}
```

Relationship Declaration Method Optional Parameters

It is possible for a relationship declaration to be modified to specify unconventional names of the relationships, clarify foreign key names, and specify the index table of a many-to-many relationship. These additional parameters may be included in any order.

Through("tableName") – This specifies the name of the index table used in a many-to-many relationship. It must be declared on both sides of the relationship and can only be used in a HasMany relationship declaration method.

Table("tableName") – This is used to specify the actual database table name if the relationship is given a name that differs from convention. It can be used on any of the relationship declaration methods.

Key("columnName") - This is used to specify the appropriate name of the foreign key when there is more than one relationship between objects. It can be used on any of the relationship declaration methods.

Usage:

```
class Person extends Core{
    public function __construct(){
        $this->HasOne('address', new Table('employee_addresses'));
        $this->HasOne('computer', new Key('laptop_id'));
        $this->HasMany('checks', new Through('check_assignments'));
    }
}
```

Conventions

Core makes assumptions about the spelling and existence of tables and fields based on the following conventions:

Table and Field Names – Table names should be all lowercase with underscores separating words.

Model Class Name - The class name of the model should be the same as the name of the table it talks about except singular and in camel-caps with the first letter capitalized.

Relationship Names – Relationship names should be the same as table names by convention. The only exception is that singular relationships (HasOne, BelongsTo) should have the table name in the singular.

Object Attributes - Attributes of objects including their relationships are in camel-caps and are only plural in a HasMany relationship.