Much of the work was largely "cut and paste," but if that's all a human is doing, then maybe it is possible to get a computer to do it.

Field	Type	Attributes	Null	Default	Extra
id	int(11)		No		auto_increment
first_name	varchar(50)		Yes	*NULL*	
last_name	varchar(50)		Yes	*NULL*	
company_name	varchar(50)		Yes	*NULL*	
address	varchar(60)		Yes	*NULL*	
city	varchar(40)		Yes	*NULL*	
county	varchar(50)		Yes	*NULL*	
state	char(2)		Yes	*NULL*	
zip_code	varchar(10)		Yes	*NULL*	
phone	varchar(15)		Yes	*NULL*	
fax	varchar(15)		Yes	*NULL*	
email	varchar(50)		Yes	*NULL*	
occupation	varchar(50)		Yes	*NULL*	
employer	varchar(50)		Yes	*NULL*	
card_number	varchar(4)		Yes	*NULL*	
amount	decimal(10,0)		Yes	*NULL*	
response	varchar(10)		Yes	*NULL*	
authorization	varchar(6)		Yes	*NULL*	
trans_id	int(11)		Yes	*NULL*	
trans_time	timestamp(14)		Yes	*NULL*	
reference	varchar(20)		Yes	*NULL*	

Figure 1 - Donations Table Design

One of the greatest things about the architecture of SQL databases is that the metadata (the data describing the database and its content), is stored in an SQL database. Simple queries with results like those shown in Figure 2 make it easy to introspect (literally "look at oneself") the tables in the database and the contents of those tables. Given that the metadata is available to programs in general and to PHP in particular, it is relatively easy to write programs to process database table structures in a very general fashion. Table names, fields within tables, data types of fields, use of fields as keys, and so forth are all available for processing.

Field	Type	Null	Key	Default	Extra
id	int(11)		PRI	NULL	auto_increment
first_name	varchar(50)	YES		NULL	
last_name	varchar(50)	YES		NULL	
company_name	varchar(50)	YES		NULL	
address	varchar(60)	YES		NULL	
city	varchar(40)	YES		NULL	
county	varchar(50)	YES		NULL	
state	char(2)	YES		NULL	
zip_code	varchar(10)	YES		NULL	
phone	varchar(15)	YES		NULL	
fax	varchar(15)	YES		NULL	
email	varchar(50)	YES		NULL	
occupation	varchar(50)	YES		NULL	
employer	varchar(50)	YES		NULL	
card_number	varchar(4)	YES		NULL	
amount	decimal(10,0)	YES		NULL	
response	varchar(10)	YES		NULL	
authorization	varchar(6)	YES		NULL	
trans_id	int(11)	YES		NULL	
trans_time	timestamp(14)	YES		NULL	
reference	varchar(20)	YES		NULL	

Figure 2 - Table Description Query

Once I realized that this data was available, I quickly designed and wrote a simple PHP program to generate classes from the metadata of a MySQL table. A day later, I had a program, buildClass.php, which reads the metadata of a MySQL table in a database and emits a class derived from SQLData that provides the framework for manipulating data within a single table of a database. Example 1 is a partial listing of the generated class included here for discussion of the generated classes.

```
<?php

//
// Class: Example
// Table: example
// Database: APG
//
// Generated by buildClass.php, written by Dick Munroe
(munroe@csworks.com)
//

include_once("SQLData/class.SQLData.php") ;                    // (1)
include_once("SDD/class.SDD.php") ;

class Example extends SQLData                                  // (2)
{

    //
    // Private (or constant) varibles.
    //

    var $m__tableName = 'example' ;                            // (3)

    //
    // Constructor
    //

    function Example($_dataBase,                               //(4)
```



```
                    $_host="localhost",
                    $_login="",
                    $_password="")
    {
        $this->SQLData($this->m__tableName, $_dataBase, $_host,
$_login, $_password) ;
    }

    //
    // Accessor Functions
    //

    function setId($theValue)                                    // (5)
    {
        $this->set('id', $theValue) ;
    }

    function getId()                                             // (6)
    {
        return $this->get('id') ;
    }

    function initId($theValue)                                   // (7)
    {
        $this->init('id', $theValue) ;
    }

    function un_setId()                                          // (8)
    {
      $this->un_set('id') ;
    }

    function is_setId()                                          // (9)
    {
      return $this->is_set('id') ;
    }

    //
    // Default update selector
    //

    function needUpdateSelector()                                // (10)
    {
        if (($this->is_setId()))
           return "where `id` = '" . $this->escape_string($this-
>getId()) .
                  "'" ;

        trigger_error("Internal Logic Error: No key data present",
                      E_USER_ERROR) ;
    }

    //
    // Insert function
    //

    function insert()                                            // (11)
    {
        $theReturnValue = parent::insert() ;
        if ($theReturnValue)
        {
```

```
            $this->initId($this->fetchLastInsertId()) ;
        }
        return $theReturnValue ;
    }

    //
    // Debugging Functions
    //

    function print_r()                                          // (12)
    {
      $sdd = new SDD() ;
      print($sdd->dump($this)) ;
    }
}

?>
```

Example 1- Automatically Generated Example Class

1. The various underlying components of the application generation, in particular the SQLData class from which all specific table classes are derived and the Structured Data Dumper class which is used. To make the code generated by the programs referred to here, SQLData and SDD must be installed in your PHP include path. See the PHP documentation for details.
2. Every table specific class is derived from SQLData, a class supporting generic table data storage and updating. Essentially, the table specific classes are convenience classes to make dealing with specific tables easier. Note that the first character of the table name is upper cased to make the class name.
3. This provides the binding between this class and the underlying MySQL table.
4. The constructor for the table-specific class. Since the table name is wired into the class, the remainder of the MySQL database access information must be provided when the class is instantiated. No data oriented constructors are provided, as most of the table-specific class data initialization occurs either as a side effect of accessing the table through the underlying SQLData interfaces or from interactions with a user through web forms.
5. set* member functions set the named field to a value and note that the value is now "dirty" and should be flushed to the database when the next update or insert operation is issued. A set* member function will be created for each field in the table metadata.
6. get* member functions get the named field value and return it to the caller. A get* member function will be created for each field in the table metadata.
7. init* member functions are the same as set* functions but the data is not set as "dirty" and will not be flushed to the database when the next update or insert operation is issued. An init* member function will be created for each field in the table metadata.
8. un_set* member functions delete data for a field from the underlying SQLData class. Once deleted, the field data is no longer considered in insert or update operations. An un_set* member function will be created for each field in the table metadata.
9. is_set* member functions are predicates returning true if the field has data in the underlying SQLData class. Data is stored for a field using either the set* or init* member functions. An is_set* member function will be created for each field in the table metadata.
10. The insert and update member functions (available using SQLData) both take an optional selector to indicate which record in the table should be modified. For properly designed tables with keys, it is generally possible to provide a set of default selectors, depending on which keys in the table currently have data associated with them, to be used when a selector is not

- Syntactic Validation
- Semantic Validation

User Interface

As shown in Figure 2, the table metadata has the basics, field name, data type, size of data and whether data is required (not null). Given this information, it was easy to write another program much like the buildClass.php program to construct a simple user interface using HTML forms to display and capture data and to link that interface to the client-side syntactic validation framework and to the server-side semantic validation.

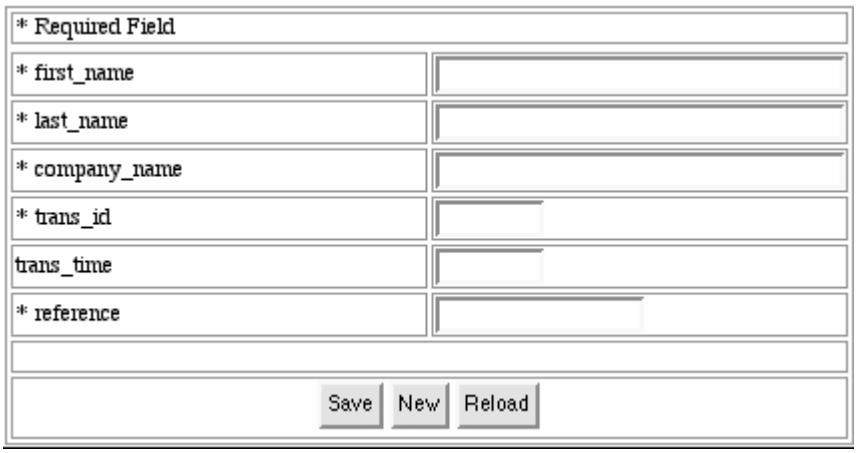

Figure 3 - Generated User Interface

Figure 3 has been edited for space reasons and shows part of the user interface generated for the Example table.

It is important to remember that my goal for the FDP was not to produce a completely polished and fully functional web application solely from MySQL metadata. It was only to produce something that, with not much effort, could be turned into a "completely polished" and fully functional web application.

```
<link rel="stylesheet" type="text/css"
href="syntacticValidationFramework.css" />              (1)
<script type="text/javascript"
src="syntacticValidationFramework.js"></script>         (1)
<script type="text/javascript"
 src="form.Example.js"></script>                        (1)

<form name=data method=post action="process.Example.php"
 onSubmit="return validate(this) ;">                    (2)
<table name=dataTable id=dataTable border=1 cols=2>
        <tr>
            <td colspan=2>
                * Required Field
            </td>
        </tr>

        <tr id=errorRow>                                (3)
        </tr>
        <tr>
            <td><span id="spanFirst_name" class="inline">*
                 first_name</span></td>
            <td>
<input name="first_name" id="first_name" type=text size=40
maxlength=50 required="first_name is required" validate="return
```

```
validateFirst_name(what)" value="">           </td>
                (4)
        </tr>
<tr>
            <td><span id="spanTrans_time"
                 class="inline">trans_time</span></td>
            <td>
<input name="trans_time" id="trans_time" type=text size=10
maxlength=10 required="" validate="return validateTrans_time(what)"
value="">           </td>
   (5)

        </tr>
<tr>
            <td align=center colspan=2>                  (6)
                <button name=save id=save type=submit>Save</button>
                <button name=new id=new type=button
onClick="window.location.search='?action=new'">New</button>
                <button name=reload id=reload type=button
onClick="window.location.search='?action=reload'">Reload</button>

            </td>
        </tr
    </table>
</form>
```

Example 2 - Generated HTML

A quick look at the HTML generated by buildForm.php (Example 2) is instructive.

1. These are the hooks to the "syntactic validation framework" discussed more fully in the next section. Unless buildSyntacticValidation.php has been run before buildForm.php the JavaScript components of the syntactic validation will not be included.
2. This is where the syntactic validation framework is actually invoked. When a Submit button is clicked, the onSubmit code is called. The form is not actually submitted unless or until all syntactic errors have been corrected. If buildSyntacticValidation.php has not been run, the onSubmit code is omitted and no syntactic validation will be done.
3. Another hook for the syntactic validation framework. This row is where the error information (if any) will be displayed by the framework.
4. This is a typical required form field. Validation for the field is provided by the validation attribute. Note the required attribute for required fields is non-null.
5. This is a typical optional form field. Validation is still required (the validation may always succeed, of course) and is done for consistency. Note that the required attribute is the null string for optional fields.
6. The action portion of the user interface. **Save** causes the captured data to be (optionally) syntactically validated and sent to the server for further processing. **New** clears the form and starts the data capture process over. **Reload** discards any data changes and starts the data capture process over.

Syntactic Validation

The syntactic validation requires JavaScript (now known as ECMAScript). Most modern browsers support JavaScript so this requirement isn't all that restrictive, only leaving out text-only browsers such as lynx or links. It relies on the Document Object Model (DOM), defined by the World Wide Web Consortium. Unfortunately, Microsoft's Internet Explorer is not particularly compliant with the DOM. It was possible to design an adequate web-browser independent framework providing primitives to handle collection and displaying of error data, validation of required fields, and a driver to validate a form upon submission.

Since my goal was to generate most, but not all, of a web-enabled application automatically, the validation hooks had to be associated with the individual fields rather than generated monolithically. Further, not every form would necessarily need syntactic validation.

This is handled by running (or not running) buildSyntacticValidation.php. This program generates the JavaScript routines to do the syntactic validation for each field in the form. If buildSyntacticValidation.php is not run, when the form is generated by buildForm.php no syntactic validation will occur at the time the form is submitted.

The syntactic validation framework is provided in a separate JavaScript file, syntacticValidationFramework.js. Understanding the details of the DOM that allow the framework to work is left as an exercise to the reader.

```
<form name=data method=post action="process.Example.php"
 onSubmit="return validate(this) ;">
```

Example 3 - Syntactic Validation Hook

Example 3 shows the hook between the user interface (form) and the syntactic validation framework. The onSubmit action is taken when a submit button is clicked. A pointer to the form object requiring validation is passed to the framework, along with the contents of the form used to actually determine what validation needs to be done. If the validation framework returns false, the form's data is not sent to the server.

```
<input
    name="first_name"                                    (1)
    id="first_name"                                      (2)
    type=text
    size=40
    maxlength=50
    required="first_name is required"                    (3)
    validate="return validateFirst_name(what)"           (4)
    value="">
```

Example 4 - Field Validation Details

The Document Object Model requires all attributes of a tag to be represented. This means that the page designer can put anything into an arbitrarily named attribute. Each field to be validated must have an unique id, an indication if the field is required, and a pointer to the routine to be used to validate the field's contents.

Example 4 shows a typical input tag with the hooks for the syntactic validation framework.

1. The name of the field received by the server during semantic validation and processing.
2. The unique id of the field. By convention it is always the same as the name field. The DOM interface is most easily used if each tag has an id by which the field can be found.
3. If the field in the database must have a value, the required field must be non-null and must contain the message to be presented to the user should the contents of the field be omitted.
4. Invocation of the validation framework routine for the field.


```
function validateFirst_name(what)
{
    //
    // Validate first_name field value
    //

    if (!isRequired(what))
    {
        return false ;
    }

    return true ;
}
```

Example 5 - Template Validation Routine

BuildSyntacticValidation.php generates a JavaScript routine identical in all but name for each field in the user interface. The validation framework calls the validation routines with a pointer to the field to be validated. Each validation routine must return either false (validation failed) or true (validation succeeded) and prepare any error text to be displayed at the end of validation.

The current syntactic validation framework can easily be modified to fit a variety of error reporting and interaction styles without modification to any of the generation code. Or something completely different can be written to meet site-specific requirements. After all, the generating code is in the public domain.

Semantic Validation

The data hits the database in semantic validation. Assuming that the form passes syntactic validation when the user presses the Save (Submit) button, the contents of the form are sent in HTTP format to the server. For the purposes of the FDP, semantic validation basically took the data from the form and put it in the databases.

The semantic validation code is generated by buildProcessForm.php. The semantic validation does not happen by default. As generated by buildProcessForm.php data was either inserted into or updated in the database and then control returned to the user for further work. Any semantic validation was considered to be custom code and to be written by hand. This is consistent with my goal of generating most but not all of the application.

The Complete Process

At this point, we had a set of tools that allowed the FDP applications to follow a very well defined data driven software development process. All of these tools were either free software, shareware, or easily developed in-house. The tools and process were:

1. phpMyAdmin	Design the database schema for the application. Database design was frequently a group interactive event, starting on a blackboard and quickly moving to a web browser or terminal. Only one database design only took more than an hour from discussion to completed design.
2. buildDatabaseConf.php	Generate the configuration file containing the information necessary to access the MySQL database.
3. buildClass.php	Create the PHP class to make it easier to manipulate the data. All the PHP code generated by the build* programs use the generated class.
4. buildSyntacticValidation	Create the individual field validation routines for the syntactic validation framework.
5. buildForm	Create the user interface.
6. buildProcessForm	Create the PHP code to store/update data in the

	database.
7. Apache, FreeBSD	Install the generated application on the development webserver.
8. Netscape, Internet Explorer, MySQL	Test the user interface/database interaction.
9. emacs, UltraEdit, Putty, WinSCP	Modify the generated application to full function and integrate with the production FDP web site.

Using this process, you can go from the completion of a database design to a prototype application in a matter of minutes.

Conclusion

The techniques associated with automatic program generation are well understood and widely used in many application generators on any number of platforms. By spending a few days writing tools, I made it much faster and easier to turn out finished applications. Experience showed that about 80% of the finished application can be automatically generated. Finished applications, fully integrated with the production FDP web site, are frequently produced in a day, with the vast majority completed in less than two days.

In all likelihood, similar productivity levels can be achieved elsewhere with site-specific versions of the tools described here.

All code developed for the FDP discussed in this paper is available from www.phpclasses.org. You will have to join www.phpclasses.org in order to download the code, but the membership is free. PHP is available from www.php.net and MySQL from www.mysql.com. These can be built and run on UNIX, Linux, and Windows platforms. Cygwin is available from www.cygwin.com if you want a UNIX-like environment for your Windows PC. If you're going to do any serious MySQL database development phpMyAdmin is a must and can be downloaded from www.phpMyAdmin.net. All are free software.

Acknowledgements

I'd like to thank Chris Sands, the director of IT for the Florida Democratic Party, for permission to publish this work. Many thanks to all the folks who supported the FDP during the 2004 campaign, lots of nights wouldn't have been possible without the free Diet Coke and junk food. I'd also like to thank Manuel Lemos, the creator of www.phpclasses.org. My job at the FDP would have been substantially more difficult if his site didn't exist. Last, but definitely not least, thanks to the many unnamed developers who have put together so much fine software and put it into the public domain and in particular to those folks who have contributed to PHP, MySQL, Apache, Cygwin, and FreeBSD. Without it the work we did in Florida during the last presidential campaign would have been impossible.

For more information

If you need more information about the work described here, contact Dick Munroe at munroe@csworks.com