# EPCC-SS99-04

# Engineering Learning on the Web

**Robert Byrne**

## Abstract

Last year a significant part of a first year undergraduate Physics half-course, run at Edinburgh University, was ported to use the web to supplement both teaching and assessment. From the start a deliberate attempt was made to decouple the source material produced from the actual web delivery mechanism, which in this case consisted of the semi-commercial WebCT package, thus providing a certain degree of independence from any one particular delivery mechanism. Part of this entailed the construction of a question-authoring CGI (Common Gateway Interface) script. This would enable a course team to author questions via the web. The database of questions thus produced could be exported to WebCT and also LaTeX, to provide a paper-based version of the material. However in its development the authoring script became what can best be described as a *monolithic working prototype*. The aim of this project was to streamline and modularise the existing code, remove operating system specifics, add new functionality, examine security issues and provide documentation. The result of all this produced a more manageable piece of code that should be easier to maintain and extend in the future.

## Contents

# 1 Introduction

## 1.1 Background

The *authoring* script that was to act as the basis for this project was realised as part of a collaboration between the University of Edinburgh's Department of Physics and Astronomy, SELLIC[1] and EPCC. The object was to port a significant part of the first year physics half course, *The Foundations of Physics*, to use the web to supplement lectures and provide a means for self and course assessment for students. As time was short an already established course delivery package was chosen. This was WebCT[2]. WebCT started off as an academic project at the University of British Columbia, Canada but moved on to a semi-commercial status in order to provide sufficient funding to develop it further.

Despite the fact that WebCT has its own course generation tools a deliberate attempt was made to decouple the creation of the source material intended for the course and the actual delivery mechanism. This would safeguard the large investment in time and effort that would be required to produce the course material from any weaknesses in the delivery agent or the continuity of the company behind its production[3]. It is hoped that newly emerging standards like IMS[4] will, in the future, make the necessity to take such safeguards a thing of the past.

The purpose of the authoring script was thus relatively simple: to allow lecturers to write course questions that would later be exported to WebCT (or some other course delivery mechanism). As previously stated WebCT already comes with its own course authoring software but as LaTeX was to be used as the underlying text-based source some preprocessing would be necessary before questions could be exported to WebCT. It would thus not have been possible to enter the questions directly into WebCT with this choice of input format, even if de-coupling question authoring from the system was not an aim of the project.

WebCT itself comes with a whole host of different types of *quizzes*, e.g. text entry boxes, multiple choice, etc., that can be used for course and self-assessment but the type that was used by Physics was restricted to *multiple choice* type questions. A text based upload format can be used to upload the questions into WebCT and this is what the authoring system outputted after all the preprocessing.

One of the design weaknesses in the original version of the authoring script was that what was required was not entirely clear at the start, so, in effect, the script itself became a working prototype. As functionality was added (and taken away) it became apparent that new features would be required or desired. Bug fixes, through time constraints, became hacks and the script, although working, became monolithic and unwieldy. Maintainability was becoming precarious and difficult. Also, although originally intended to allow some independence from WebCT by being able to export to other potential delivery mechanisms, the necessity for rapid development had resulted in code that was too tightly coupled to WebCT's own input format, and it would be difficult to disentangle this with the existing version of the code. Thus, with the benefit of hindsight it was now time to put some form of design back into the project. This, in theory,

[1]SELLIC stands for *The Science and Engineering Library Learning and Information Centre*, for more details of this project see: http://www.sellic.ed.ac.uk/.

[2]For more information on WebCT see: http://www.webct.com/.

[3]This is not to say that WebCT was not up to the task, but rather a safeguard in case it did not come up to expectations, or a better alternative came along.

[4]For more information on IMS (formerly *Instructional Management Systems*) see: http://www.imsproject.org/.

would make the code easier to maintain and expandable in the future.

## 2 The original script

### 2.1 Overview

The general principle behind the design of the script in itself was simple. A root directory would be used to store the database. Within this a set of subdirectories would store question *categories*, and within each *category* subdirectory, the individual questions would be stored in their own individual subdirectories. This structure was mapped to a set of pull-down menus, viewed via a web browser. Starting from the top, a set of pull down menus would allow a category to be chosen (creation and deletion of existing categories could also be performed from the top level).

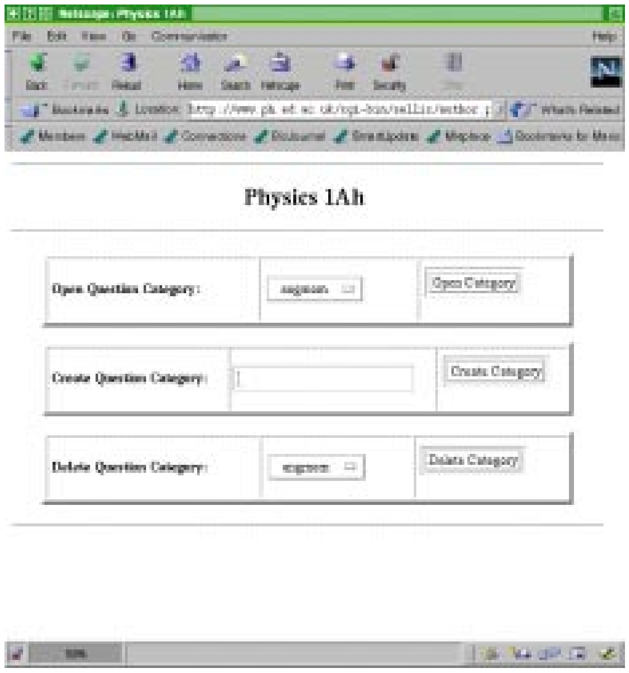

Figure 1: Choosing a question category.

The questions themselves are stored in plain text in a marked-up fashion. This would make conversion of the questions to a generalised format, such as XML, quite easy and should make porting to a future IMS compliant format relatively trivial. There is an issue as to the efficiency of the system once the questions database becomes very large but up to the present point in time this has not become an issue other than the fact that the export mechanism becomes very slow. The main reason for this is that latex2html is used to convert the question files into HTML and this process is very slow. It was not possible, within the time available, to come up with an in-house LaTeX to HTML converter. (Getting a small speed increase at the cost of re-inventing the wheel is likely not worth it anyway, but possibly worth considering.)

Once a question category is identified or created, each question stored in the database can be manipulated at an individual level, i.e edited, previewed, deleted, etc.; or collectively where a subset of questions or the entire category can be exported to WebCT. This would pack up the processed files into a zip archive that could be uploaded into the WebCT framework. The preview mechanism only works once a user initiates the export mechanism to WebCT, whereby the preview files are created. This gives a representation of what the questions would look like when viewed under WebCT.

### 2.2 Security model

Clearly security is a major concern – one does not want students to have a preview of questions that may be used under exam conditions. To access the question database one has to go through a password protected page. Once successfully done a cookie is set up that prevents entry to the remainder of the script. Additionally only a subset of allowed hosts can use the script. These machines are ones that students will not have access to. Normal Unix file permissions safeguard the directories in which the questions are stored.

## 3 Issues arising from the existing code

### 3.1 Perceived problems with the original code

The original Perl version of the authoring script consisted of some 4000+ lines contained within one file. Within this some 27 subroutines were to be found alongside pieces of embedded JavaScript. Clearly the existing set up had its disadvantages:

- As all the code was to be found within one large file the entire authoring script had to be compiled whenever a user required a task performed, however small.
- Lack of modularity
  - The size of the script and its evolution made the coupling between the different subroutines fuzzy. A change in one could potentially have unforeseen consequences in another thus compromising the maintainability of the code. For modularity, Perl recommends the use of modules – usually single files with their own variable namespace, similar to an object in object-oriented programming styles. This idea was examined but modules were not used in this project as the requirements did not call for it. Instead the new form envisaged for the system was to split the existing script into a large number of smaller separate scripts.
  - Revision management had to be applied to the entire file, instead of the relevant components of the system. This would mean that, over time, changes made to various parts of the system would become coupled to each other, so that going back to a previous revision to remove or change some code would mean losing worthwhile changes in other areas. The bigger a single revisable unit is, the more revision coupling affects the usefulness of revisions.
- Perl can be a particularly bad offender when it comes to obsfucation (unreadability), particularly with its use of regular expression matching (although this is where a lot of its power lies too). For this reason care should always be taken to make a program readable, for the sake of future maintainability. The project therefore aimed to make the code as easy to understand as possible by choosing appropriate variable and subroutine names,

and laying out the code in an easier to read fashion (and removing redundant code and commented-out “dead code”). Comments embedded within the script were not always helpful and even sometimes misleading having been left there from some previous incarnation.

There were inefficiencies in the code, some of which were flagged for future changes, such as a large block of repeated code, which to be removed would require some large scale re-working of the code; other smaller problems were easily fixable, such as Perl regular expressions with redundant elements, or sections of conditional code that could be written more concisely.

It was also seen as worthwhile to remove operating system specifics (i.e. using a Unix shell to perform various tasks like removing, moving (or renaming) files, etc.) potentially making the script portable to other operating systems.

### 3.2 Solutions

The first task at hand was to split the existing script into a number of smaller scripts according to the functionality. This would improve on the large file compilation problem, and the modularity by making each file focus on the routine at hand and thus make changes easier to make. As a side issue it would considerably reduce revision coupling – files could be stored in separate RCS or CVS revision archives.

The code itself was streamlined and re-written in sections. Re-naming of variables and subroutines was trivial, although there is the issue of “semantic propagation” of variable names (see §3.3).

The opportunity was also taken to rationalise and improve on the existing comments.

### 3.3 Discussion: re-engineering code

**Making the Global Local:** Some global variables occur throughout a program, but are re-defined when they are used in a new subroutine. The structure of the authoring script was such that, although most of the variables were global, their use was local. This eased the process of splitting up a script which at first glance appeared to be filled with troublesome global variables.

**Semantic propagation of new variable names:** Although the same variable name may appear later in a script, it does not necessarily have the same meaning. When a more generic name is made specific and all entries throughout the script are changed, the name can change from ambiguous to misleading. The obvious solution is to re-evaluate variable meanings on a subroutine by subroutine basis. This has been done as much as possible.

## 4 Method

### 4.1 Details of the approach taken

In the original design the authoring script, `author.pl`, was called with a set of parameters, the first of which always specified which subroutine was to be used to perform the required task. In the new design the function call now refers to a separate script based on the original subroutines. The subroutine/script names have been slightly changed to make their functionality clearer, and their names more readable. Although some of the subroutines in the original code called each other within the script, e.g. the deletion of a question then redirected you to the listing of questions within the same category, it was not necessary to include both subroutines in the same script as the calling mechanism could easily be changed so that the client requested a different script. This was done using the "`Location:` " http directive and, in some cases, a "`meta-refresh`" was used to redirect the browser to the desired end location. The end result of this was to improve the modularity of the code.

However, some variables and subroutines were required or used by the whole package. These were thus placed in a Perl file, `authorGlobal.pl`, and included using the Perl "`require`" command. (See the appendix for a complete list of all scripts and variables, with explanations.) The content and purpose of this file is very similar to the code that was executed at the beginning of the old authoring script.

## 4.2 Differences between old and new authoring systems

It should be emphasised that despite the improvements the code of the new authoring system as a whole works in a very similar fashion to what was there before. Here are the changes:

- Instead of calling the same script every time a task is required, a specific separate script is called. This script is named using the convention `$...Script` which is a variable defined in `authorGlobal.pl`. This allows less obvious script names to be used, if this is ever required for security reasons, noting that any change in this variable name must coincide with the actual script name.
- Where before, a subroutine may have lead to another subroutine being called as its last action, it now outputs an http directive of the form "`Location: ...`". As any subroutine that did this produced no output of its own (and all subroutines assumed that nothing had been sent so far to the browser), using the `Location` directive worked well as it was part of a standard http header that the browser expects. Some of the subroutines that are commonly called from several other functions have been included in the script `authorGlobal.pl`.
- Whereas before adding a new feature would have required the addition of yet another subroutine with the associate code to a switch-style statement near the top of `author.pl`, now one has to:
  - Create a new script.
  - Give it the standard header that is contained in all the other scripts. This header does the following:
    - ∗ sets a variable giving the physical location of all scripts (this could be moved to `authorGlobal.pl`, unless that is the scripts are to be kept in different directories);
    - ∗ sets a variable to the script's own filename; and uses "`require`" to execute `authorGlobal.pl`, which does necessary tasks such as checking that the client is one of the machines allowed to access the script.

- Note that:
  - CGI data obtained through the `POST` method is made available to the remaining scripts using a `%FORM` hash-array which is defined and initialised in `authorGlobal.pl`. See the Glossary for an explanation of CGI POST data.
  - CGI data obtained through the `GET` method must be parsed by the scripts themselves. This is an inherited feature of the original system, and could be moved to `authorGlobal.pl` by a future maintainer. Again, see the Glossary for an explanation.

- The hosts which are allowed access to the system are now listed in a separate file: `allowedHosts.pl`. (See the appendix for a complete list of scripts and subroutines, with explanations.) This will allow modification of the access list without having to edit the main scripts.
- The user interface looks a little prettier.

## 5 Conclusion

### 5.1 Project goals

Many of the initial project goals were achieved: the code was streamlined and modularised, all Unix operating system specifics were removed, some new functionality was added and security issues were examined, although it was decided that no new security measures should be added at this time. See below for a future security measure, regarding the password cookie, that could be implemented.

### 5.2 Future directions for the authoring system

There is still some rationalisation to be done and some changes in the way the code works internally to be made:

- The collection of CGI GET data could be performed in `authorGlobal.pl`. (See the Glossary for an explanation of CGI GET data.)
- Improve security with the password cookie – a good idea would be to have a constantly changing code, based on, for example, the time the user logged-in. That this code is currently applicable should be noted at the server end.
- De-couple the LaTeX and HTML output features, and allow HTML only, and LaTeX and HTML together export features.
- Fix some small remaining bugs such as a path output problem when diagrams cannot be found.
- The database and LaTeX files have extra blank lines added every time they are edited. This should be fixed.

The obvious future changes would of course involve adding to the functionality of the system. Specifically:

- A new sub category structure for questions.
- A one-stop edit facility for question status.
- A counter for the number of questions.
- Show last modified date of a question.
- A web interface for the creation of a student tracking database.

I'm from Dublin, Ireland and have lived there for all of my life. I am studying Computer Science in Trinity College, University of Dublin. I'm interested in science, technology, music and computers, and would consider careers in any of these areas! I tend to fill my spare time with music, such as playing in groups, studying classical piano, and attending Jazz classes during term time in college.

Supervisor: Dr. Mario Antonioletti

## 6 Glossary

**Tainting** Perl has a tainting system in which any data from the outside world is not allowed affect the outside world (usually through a disk I/O operation) unless it is checked with a regular expression. Taint checking is automatically turned on when the suid or sgid bits are set on the Perl script, as they are for all scripts in this system. Taint checking can be set manually with a -T parameter on the Perl interpreter's command line.

**suid and sgid bits** "Set User IDentification" and "Set Group IDentification". In Unix an executable file can have one or both of these bits set. They allow the executing file file access privileges that the *owner* (whether user or group) of the file has, even if the user running the file does not have these privileges. In this system it allows the World-Wide Web user, through CGI (see below), to make changes to data on the server that doesn't belong to him or her.

**CGI** Common Gateway Interface. This is a protocol that allows executable programs on a web server to interact directly with a remote World-Wide Web user. There are a number of standard mechanisms to allow a remote user to send, via the web, command-line style parameters to the script, and to read its output, usually in HTML. It is also possible for a web browser to instruct the server to set environment variables in the executable's environment. This is often used to transfer HTML FORM data.

**CGI GET data** The GET method entails passing information together with the URL of a CGI script, similar in operation to a command-line parameter, and in fact all GET data arrives at the CGI script as the first and only command-line parameter. There are standard libraries to handle CGI GET data, but in this system the data are parsed manually. The GET data is gathered by each individual script.

**CGI POST data** The POST method entails passing information through HTML FORMs (usually). These are passed to the CGI script at the server end as environment variables. All CGI POST data are gathered by the `authorGlobal.pl` script, which is included (and hence executed) by every script.

**HTML** HyperText Mark-up Language. A mark-up language is a set of *tag definitions* which allow descriptions ("meta-information") of parts of an electronic document to be embedded in the document. HTML is used as the standard document format for the World-Wide Web.

## 7 Appendix: details of scripts and subroutines

Here is an explanation of every script in the system, along with the subroutines they contain.

### 7.1 Script details

- `AddQuestion.pl`

  Adds questions to the existing database files. Writes them to disk. Once written, reads in the file and displays it.

- `allowedHosts.pl`

  This file stores the names of the hosts which are allowed access to the system. The decoupling of this from the main body of the script allows the access list to be edited by externals without having to go into the scripts to make modifications.

- `authorGlobal.pl`

  This file stores global variables and subroutines and performs the initialisation of data for all scripts in the system. Specifically it:

  - Includes the libraries needed by the system with "`use`" – `File::Path` and `File::Copy`.
  - Stores the passwords for access, and for deleting questions (these are encrypted).
  - Stores directory definitions.
  - Defines all of the internal tag format variables.
  - Defines all of the actual script names with $...Script variables.
  - Checks that the accessing host is allowed (it uses "`require`" to include `allowedHosts.pl`).
  - Checks that the script running it has the *suid* and *sgid* bits set. (See the end of the report for a Glossary of terms.)
  - Gets the CGI POST data from the environment.
  - Defines the subroutines listed below in the Subroutine section next to `authorGlobal.pl`.

- `CanvasPage.pl`

  Outputs a start-up blank page that is overwritten by the question layout editor (`$QuestionEditorLayoutScript`, see `authorGlobal.pl` for definitions of $...Script variables) as soon as the first question is inputted.

- `CreateCategory.pl`

  Creates the directory in which a question category will go. Creates content for an exercise.

- `DeleteCategory.pl`

  Deletes all the contents under a category subdirectory.

- `DeleteQuestion.pl`

  Allows a single question to be deleted form the authoring script database.

- `DeleteQuestionSubset.pl`

  Allows a subset of questions to be deleted from the authoring script database.

- `EditFrame.pl`

  Creates frames to be used in the editing of questions.

- `EditQuestion.pl`

  Allows editing/updating of an existing question.

- `Export2WebCT.pl`

  Exports questions to WebCT format, latex and html.

- `IncorrectPassword.pl`

  Gives appropriate output for a failed password entry.

- `ListCategories.pl`

  Allows the creation and editing of existing question categories; can also delete entries.

- `ListQuestions.pl`

  Lists all the questions stored in the database of a given category. It allows editing of stored solutions or the exporting of a subset or the entire database into a format that is uploadable by WebCT.

- `PasswordCheck.pl`

  Checks the password entered and sets a cookie if it is the correct password.

- `QuestionEditorLayout.pl`

  Controls the behaviour of the question input window. All the output from this window dictates what goes in the Canvas: see the script `$CanvasPageScript`.

- `RenameQuestion.pl`

  Renames a question.

- `ScriptPage.pl`

  Outputs a small frame used to contain some of the JavaScript that controls the question layout editor (see `$QuestionEditorLayoutScript`).

- `TopPage.pl`

  Gives the user a password check for entry to the rest of the system. A cookie is set upon the correct password being entered (by `$PasswordCheckScript`).

- `ViewQuestion.pl`

  Creates a page that can then be used to redirect to the preview of a question.

## 7.2 Subroutine details

- `Export2WebCT.pl`: Initialisation

  Along with getting the CGI GET data, this subroutine is mostly concerned with setting up some files for historical reasons, and will probably be removed.

- `Export2WebCT.pl`: `CreateLatexFiles`

  This generates LaTeX both for printing and for future embedding into a WebCT input format file.

- `Export2WebCT.pl`: `CreateWebCTQuiz`

  Uses latex2html to convert the latex created in the previous subroutine into HTML, for embedding in a WebCT input file.

- `Export2WebCT.pl`: `CreatePreview`

  This is the reverse process for previewing. The WebCT format produced in the previous subroutine is converted to HTML.

- `Export2WebCT.pl`: `CreateSelfTest`

  Creates the WebCT self test data, which must be in a slightly different format to the WebCT question upload format.

- `authorGlobal.pl`: `doc_head`

  Writes a HTML document header, including a http preamble. Includes a parameter for a document title.

- `authorGlobal.pl`: `doc_foot`

  Writes a HTML document footer.

- `authorGlobal.pl`: `IsCookieSet`

  Reads the cookie information from the CGI environment and if the password cookie is not set, outputs an error message. The cookie information remains available to other scripts, assuming the password is set. Every script runs this except for the entry script – `TopPage.pl`.

- `authorGlobal.pl`: `ProgramHalt`

  A standard “bomb-out” routine that halts operation outputting an optional error message which can be passed as a parameter.

- `authorGlobal.pl`: `ReadTagFormat`

  A routine to read the question data from file.

- `authorGlobal.pl`: `ReadEntriesInDirectory`

  Reads a given category directory and extracts the names of questions.

- `authorGlobal.pl`: `RemoveTree`

  Attempts to remove all files and directories passed to it.

- `authorGlobal.pl`: `RecursiveRemove`

  Used by RemoveTree to traverse recursively a directory structure, destroying it as it goes. *Untaints* directory information it gets.

- `authorGlobal.pl`: `Untaint`

  Untaints every variable passed to it: every element of a list. See the Glossary for an explanation of tainting.