Curriculorum: A Computer Science Approach to Curriculum Management

Robert R. Lewis
School of Electrical Engineering and Computer Science
Washington State University [1]

Abstract

Having been assigned part-time to an administrative position (concurrently with normal academic duties) in charge of curriculum management, the author has developed *curriculorum,* a system to assist in that task that was, is, and will continue to be developed by and for a computer scientist. We describe the design of the system and how it has evolved to meet requirements that were only dimly perceived when the project started. As such, the project has been a gratifying exercise in the development of its designer's skills in data management, in software engineering, and in Python, its implementation language. We provide guidelines for the reader embarking on a similar project.

1. Introduction

In 2005, the author was appointed Academic Program Coordinator for Computer Science at his campus. This position is something like a local Chair: he is responsible for managing the program on campus, but since the School is distributed across two campuses, he reports academically to a Director on the main campus.

One of the major responsibilities of the job is overseeing course delivery: scheduling class sessions, hiring and assigning instructors, arranging classrooms and times, and evaluating instructor performance, all on an ongoing basis. As we will show below, this presented a non-trivial data management problem.

There are existing commercial products to address this problem, but they are irrelevant. Insofar as we surveyed them, they have two traits in common: a GUI front end (usually web-based) and a database (usually relational) back end. Neither of these was particularly relevant here for the following reason: The author is a computer scientist. Dealing with data is not only something he is capable of doing, it is something he *enjoys*.

The best way to solve the curriculum management (or, indeed, any similar) problem is to make devising the solution *fun.* The definition of "fun" here was guaranteed to be different from that found (if any) in using a commercial software product: Commercial software generally hides its architecture from the end user. In this case, the end user wants not only to see the architecture, but to design and modify it. As understanding of the problem evolves, the architecture can be changed and because there is a customer base of one, such problems as backward compatibility and maintaining multiple versions don't apply.

Our goal here is not so much to describe *curriculorum* (Latin for "about

[1] contact information: 2710 University Dr.; Richland, WA; 99354; bobl@tricity.wsu.edu

courses”), the curriculum management system we have built, but to inspire the reader, who is presumably a computer scientist, to use their skills to approach similar problems in their own careers.

2. Requirements

As with any software engineering project, it was important to understand the requirements that the system was expected to fulfill. These evolved over time, so we will represent the phases that the requirements went through.

2.1. The Traditional Requirements

Maintaining a course database is a typical “Database 101” assignment or in-class example. What follows is a typical statement of the problem. To show the first step of an object-oriented analysis, we will indicate likely class names capitalized with a bold font notation:

> Represent the assignment of **Instructor**s and **Student**s to **Class**es. An **Instructor** has a name and belongs to a **Department**. A **Student** has a name and a student number. A **Class** has a name, a number, and belongs to a **Department**.

This is only the start, of course. Had the project been this easy, it could all have been done with ASCII files or a simple UNIX “*db(3)*”-style database. To be useful, it needed more.

2.2. Non-Requirements

To do away with preconceived notions, it was beneficial to review traditional requirements for systems like this that were *not* called for. There were two. The first was that there did not need to be a graphical user interface (GUI) for data entry. The end user was perfectly willing to enter information with a text editor into a collection of one or more files.

The second non-requirement was that there was no need to interface with a database system of any kind. Although our university has an extensive database (several, it turns out) to keep registration and related information, we had at best read-only access to it. Nor would it have been helpful in doing things like planning future course offerings: It has no requirement to serve *all* our needs.

2.3. Reports

It is important to understand the deliverables of any project. Since this project was designed primarily for its author's use, the main deliverable was defined to be a comprehensive printed report called a “Course Plan”. This would incorporate all data being maintained in the system and would include lists of: all undergraduate and graduate CS courses, courses required by each degree, instructors, and complete schedules past, present, and planned. Each of these should be in tabular form and include all relevant attributes.

In addition to the Course Plan, each semester additional organizations within the university require information drawn from the same data. These include the bookstore (which needs a list of courses, instructors, and contact information), payroll (which needs a list of adjunct instructors (only), and payments to be made), the campus webmaster (who needs a 2-year projection of course offerings), and the registrar (who needs a list of courses, instructors, and times for the current semester).

In addition, the information contained within the database is of interest to the Accreditation Board for Engineering and Technology (ABET) [AB08], which reviews our School every seven years. An adaptation of the Course Plan to include a seven-year retrospective would be very welcome by the reviewing committee.

A guiding principle in making the system generate all of these reports was "DRY" (Don't Repeat Yourself). Each distinct object, attribute, or relation should be entered in one and only one place. This reduces the chance of inconsistency.

2.4. Checking

The system should provide some kind of checking of the input. For example, all class names should be unique. The validity of some kinds of data may depend on how it is being used. For instance, when generating a planning report it should be acceptable for an instructor not to have been assigned, but when generating a report on a term that has just been completed, an unassigned instructor should not be acceptable.

2.5. Unexpected Requirements

So far, the requirements we've discussed are those that might be encountered in any department. As we proceeded, however, we encountered additional facts about the way things were done that the approach given in Section 2.1 did not cover. We list some of them here, using the same bold-face class convention for classes as there:

- The same class (which we will henceforth call a **Course**[2]) is often taught from year-to-year. We refer to a **Session** -- an offering in the schedule for any particular term -- as a sort of "instantiation" (in an object-oriented sense) of a **Course**.
- A **Session** usually has one **Instructor**, but may be team-taught.
- Most **Course**s are taught at the home **Institution** (Washington State, in our case), which is on the semester system, but we also need to show prerequisite **Course**s from collaborating community colleges, which are on the quarter system.
- A **Session** is usually taught and attended at our **Campus** (Tri-Cities, in our case) but may be taught (via closed-circuit TV) from another **Campus** and attended by our students, and *vice versa*.
- An **Instructor** may be an adjunct, who therefore receives reimbursement on a per-**Session** basis. (*curriculorum* does not need to manage non-adjunct pay.)
- A **Session** may be *conjoint*: it instantiates multiple **Course**s offered in the same **Timeslot** to both undergraduate and graduate students.
- A **Course** may be *co-listed*: available for credit in more than one **Department**

2 This avoids the confusion between a "class" in an educational sense and a "class" in an object-oriented programming sense.

under different course numbers.

- Special topics **Course**s all have the same course number. Their content may vary from **Session** to **Session**, or it may be repeated in distinct **Semester**s.

This list is by no means complete for our campus, much less any other. Its purpose in this discussion is to point out that when implementing a *curriculorum*-like system at another university, another list of unexpected requirements is sure to come up.

2.5. Evolving Requirements and Data Availability

Again, unlike the example cited in Section 2.1, *curriculorum* needs to keep records from year-to-year for retrospective analysis (such as the ABET report) as well as future projections. When there is a change to the schema (e.g. adding a new classification scheme for graduate courses) or to the data (e.g. adding attendance figures that weren't there before or instructor ratings that only become available at the end of class), it may be necessary to retroactively edit ("refactor" in a software engineering sense) old entries. This is where a traditional table-based GUI+OODB approach becomes hard to manage.

Owing to the dynamic nature of the data, it is necessary that the design allow for "holes," such as "to be determined" entries for instructors and timeslots. The system should check these at the appropriate time, so that, for example, a **Section** with an unassigned adjunct **Instructor** can be flagged at the time that the adjunct payroll report is being generated.

3. Implementation

Once the initial requirements were understood, it became clear that an object-oriented design was called for. We chose to use an object-oriented language (rather than an OODBMS) to prototype the ideas, and the one we were most familiar with was Python [PY08]. There turned out to be so many features in the language that helped the prototyping (e.g. defaultable keyword arguments) that it became clear that Python would do very well as the actual implementation language. There was an admitted bias here: It would probably be equally convenient to implement *curriculorum* in a language like Java, PHP, Ruby, Perl, or any similar, modern object-oriented language.

There may well be OODBMSs that could do the same job, but loading the Python modules never takes more than a couple of seconds and there's only one user, so persistent, shareable data storage was not required. In addition, none of the OODBMSs we examined made refactoring as easy as the using the language. And since everything is done in ASCII files, we can use all the features of a revision control system such as Subversion [CFP04] to prevent data loss.

A software engineering project on this scale usually requires a specification document, but since the developer and client were one and the same, it was clear that with appropriate comments (and Python encourages self-documenting code), the code itself was an adequate specification. ("DRY" again.)

There are three parts to the *curriculorum* system: the module itself, the data that describes the specific curriculum, and the required report generators.

3.1. The curriculorum Module

All classes used by curriculorum are defined in this module, but it does not instantiate any objects. In principle, this means that it could be used by another department or possibly another institution, but this would be a naïve view that would not take into account any of the requirements that led to its design. Space does not permit a detailed view of the module, but we can include the most useful part for reimplementors: the names of the classes.

At present, the curriculum-related classes in this module are: **Campus**, **Course**, **Degree**, **Department**, **GraduateArea**, **Institution**, **Instructor**, **Season**, **Semester**, **Session**, **Staff** (a singleton subclass of **Instructor**), and **Timeslot**. Notice that there was no need to create a **Student** class, probably avoiding some privacy issues.

In addition, the module defines a set of curriculum-related exceptions: conditions that may arise during checking or report generation that require manual correction in the input. (In Python, these are implemented as classes.) Their names are self-explanatory: **CourseIsNotGraduate**, **NotQualified**, **NameNotUnique**, and **ImproperPayment**.

For report generation, it was also useful to include classes to support table generation in this module. They include **Table** and **Column**. Python's approach to polymorphism was especially useful here. A **Table** associates a sequence of **Column**s with an arbitrary sequence of objects. A **Column** has a header and a function. When a **Table** is printed (i.e., converted to a string), its **Column** sequence is traversed for headers on the first row and then the object sequence is traversed, one element per row. For each such row, the element is passed to the corresponding **Column**'s function, producing a string that then gets put into that column on that row of the **Table**. This is quite robust, even allowing for the elimination of duplicate entries in vertically-adjacent columns.

At present, most reports are generated using LaTeX [KD99], but doing the same in HTML, RTF, simple ASCII, or any other document description language that allowed tables would require very few modifications. The report generators are also capable of converting the graph-related data (e.g. course prerequisites) into the "dot" format acceptable to the graph visualization package Graphviz [GV08].

3.2 Expressing the Data

Objects are instantiated using the standard Python syntax in a collection of source files arranged in a directory hierarchy. Empirically, we have found the hierarchy shown in Figure 1 to be highly adaptable for this. Most of the year-to-year data is kept under the "ay_*_*" (academic year X to X+1) hierarchy. Since much of the data is the same from year-to-year, we create a new year by hierarchically copying the "ay_*_*" directory and making changes, mostly to the schedule (in our case, "schedule.py") file.

The per-term report generators ("rpt_*") are actually symlinks to a shared directory (not shown), and common **Column** definitions (e.g. "Course Name") are also imported from a common module. Most of the per-institution data is kept under the "mySchool" hierarchy.

3.2. Report Generation

The basic operation of *curriculorum* is this: Whenever we need to generate a

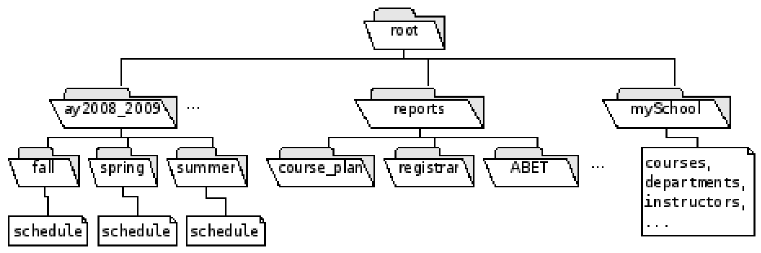

Figure 1: An Effective Directory Layout Strategy for curriculorum

report, we simply interpret the Python source for the report generator. This imports the data (actually using the Python **import** statement) which in turn imports the *curriculorum* module. The report generator then traverses the in-memory data structures to produce the report itself.

Doing this every time we need a report is justified, as the volume of data is not large enough to require a persistent database (cf. Section 2.2). As stated above, loading the data produces less than few seconds' delay on a modern desktop system.

4. Results and Future Directions

As of this writing, *curriculorum* produces a Course Plan which is 29 pages of nicely laid-out and informative PDF (the result of running the standard *pdflatex*) together with several other reports. Space does not permit us to show this, but the author would be happy to email a (slightly redacted) copy to anyone interested.

Curriculorum addresses all of our current needs. It has more than made up for the time spent for its development and maintenance in time saved during each subsequent term, and this amortization will continue. The use of a language instead of an OODBMS to implement it has had no adverse consequences.

Where it goes in the future will be determined by whatever new requests come up for the information it manages. These might include:

- recommending course tracks for specific CS specializations (e.g. games, networks, etc.)
- enrollment tracking
- additional community college transfer equivalencies
- web output (for student perusal)

Of course, another direction *curriculorum* might take is up to the reader: Design and build your own version. There is no better way for a Computer Scientist to learn their department's curriculum.

References

[AB08] *ABET*, http://www.abet.org, (as of March 10, 2008).

[CFP04] Collins-Sussman, B., Fitzpatrick B., and Pilato, C., *Version Control with Subversion*, O'Reilly, 2004.

[GV08] *Graphviz - Graph Visualization Software*, http://www.graphviz.org, (as of

3/10/08).

[KD99] Kopka, H. and Daly, P. W., *A Guide to LaTeX* (3rd. ed.), Addison-Wesley, 1999.

[PY08] *Python Programming Language*, http://www.python.org, (as of 3/10/08).