Online Evaluation in WWW-based Courseware: The QUIZIT System

by

Lúcio Cunha Tinoco

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

IN

COMPUTER SCIENCE AND APPLICATIONS

APPROVED:

_____

Roger W. Ehrich, Chair

_____ _____

Edward A. Fox Marc Abrams

January 20, 1996

Blacksburg, Virginia

Keywords: Quizzes, CBT, WBT, Evaluation, Courseware, Training

ONLINE EVALUATION IN WWW-BASED COURSEWARE: THE QUIZIT SYSTEM

Lúcio Cunha Tinoco

(ABSTRACT)

The QUIZIT system has been developed to support adaptive and standard testing, along with automatic grading, record keeping, and test administration using the WWW as a delivery vehicle. We have applied QUIZIT in connection with our NSF-supported Education Infrastructure project on Interactive Learning with a Digital Library in Computer Science to help with evaluation in a master's level course as well as a large freshmen level service course. Preliminary testing in these courses has shown us that QUIZIT is a promising supplement to other forms of evaluation in self-paced education. We also conjecture QUIZIT might be a valuable tool in distance learning environment

I wish to dedicate this work to Lia: for her love, trust and belief; and to Tony and Ruth, for the incredible support throughout these sometimes painful (and cold!), but extremely rewarding years of my life.

ACKNOWLEDGMENT

First of all, I am very thankful for the support of my two advisors and “gurus”, Profs. Roger W. Ehrich and Edward A. Fox in this project. My large thank you list includes Prof. Fox (again) and N. Dwight Barnette for volunteering to participate in this risky experiment with their classes; Prof. Marc Abrams for letting me present my ideas in his graduate level class; Prof. H. Rex Hartson and José Carlos Castillo for always supporting my work and for their feedback. Finally, I wish to thank Tommy Johnson for the great deal of help in maintaining and developing the system during the experiments.

TABLE OF CONTENTS:

1. Introduction

Since 1993, the Interactive Learning with a Digital Library in Computer Science (ILDLCS) project has been concerned with revitalizing education in our CS department, offering a large repository of WWW-based, online course material ("courseware") for both departmental and outside use [6]. Currently, the WWW-available courseware materials fill over 5000 pages developed for more than 30 different CS courses (Figure 1). New evaluation techniques that rely on the use of new tools for monitoring, logging, and analyzing our server's usage may not only improve the network response, but also help us understand how students access the course pages, both remotely and locally [6].

One important addition to these monitoring tools is a new interactive testing system, QUIZIT, which allows students to evaluate their knowledge and understanding of the WWW-based courseware, while receiving instant feedback no matter where they are located. By grading and recording student's performance on quizzes available online, QUIZIT provides a convenient and direct way for our faculty to interpret and evaluate a student's understanding of their courseware, which also comes handy as our department deals with the largest freshmen class ever and a steady increase of WWW courseware materials.

This work describes the motivation, design, implementation, and current use of the QUIZIT system in our department. We argue that the use of WWW-based automated testing integrated with our courseware can significantly reduce instructors' grading overhead, while providing yet another means of student evaluation, characterized by a high degree of interaction. It also is our goal to evaluate the use of the system in real

conditions and to assess the benefits of using QUIZIT to learn more about how students take tests in self-paced courses.

Figure 1: EI page at "http://ei.cs.vt.edu/"

Chapter 2 provides a comparison between traditional and electronic methods of test submission, and the importance of the latter in supporting greater interaction, especially in distance learning and self-paced courses. In addition, we discuss some of the pros and cons of using automated testing systems. Chapter 3 provides an overview of current automated evaluation systems, and explains in more detail some of the features and limitations of these systems when used with WWW-based courseware. Chapter 4 is a detailed description of the QUIZIT System, emphasizing how the different design aspects help to solve the common limitations of other testing systems. Chapter 5 describes the testing phase and the experimental use of QUIZIT in monitored conditions during a conference and in two courses in our department. Chapter 6 discusses some problems found and our plans for further revisions of the system. Finally, Chapter 7 concludes with a discussion of some pedagogical and technical concerns about the system's applicability.

2. Traditional vs. Electronic Methods of Test Submission: Supporting Greater Interaction

The long turn-around time between the student's submission of an assignment and evaluation feedback has been a continuing source of concern to instructors in distance and self-paced instruction settings [15]. In addition, researchers in education have frequently commented on the prejudicial effects of delayed feedback to the learning process [18], and how students often feel frustrated when they are forced to submit another assignment before the last one has been returned. In this Chapter, we compare different methods of assignment submission, showing that electronic submission with automatic grading can provide a high degree of interaction and instant feedback while significantly reducing grading time. We also discuss some limitations of automatic grading and common misconceptions about its applicability.

2.1 Traditional Processes: Letters, Papers and Orals

Probably the oldest and most commonly used method of test submission in distance learning environments is *conventional mail service* [18]. Typically, the examiner or instructor mails a test to each student taking a course; the student receives the test, takes it, and mails back the answers to some administrative section of the organization for processing and grading. The organization then mails the results to the examiner. The whole process takes one to five weeks on average. While this submission process is undoubtedly the slowest of all the others here described, it has the advantage of reaching a high population of users, assuming only a basic knowledge of how to write and post letters. Since *monitored conditions* (e.g., examiners present in the same time and location where the test is administered) are inherently infeasible in this case, these tests are usually open-book and without time limits.

The second traditional process described, and the most common in today's universities and schools, is *onsite paper submission*. Here we generally assume that both examiners and students are in monitored conditions (except for the case of homework assignments). Grading typically takes place after all the students submit their assignments. Grading time varies considerably with class size and assignment complexity, ranging from 3 days for short quizzes to 3 weeks for midterm and final exams on large-sized classes. In addition, these exams usually differ in context from others in which mail submission is used, most of them being closed-book with strict time limits. The reduction in turn-around time is derived exclusively from the removal of three postal journeys, as the grading method is left unchanged.

A third type of traditional submission process is by *oral examination*. In this process, there is direct verbal (and possibly gestural) communication between the examiner(s) and one or more students. It is possible to have students and instructors at different locations (e.g., set-up used in phone interviews), but monitored conditions are more commonly used instead. The advantage of this process is the high degree of interaction between students and instructors, and the potential reduction in turn-around time, since the students can learn about their grades immediately after the exam. The main problem with the applicability of these tests is in time scheduling and the long duration of the examination (i.e., total for the entire class). Further, there is no reduction in grading time (the difference is that grading may take place during the examination). For large-sized classes, oral examination is impractical unless there are several examiners.

2.2 Electronic Processes: E-Mail and Automated Testing Systems

With the growth of electronic mail (e-mail) and the improvements in networking infrastructure, electronic submission and processing of assignments has become very popular in universities throughout the country. The process usually consists of a student submitting an assignment via e-mail to the instructor, who grades the work and prepares some type of feedback in the form of comments or annotations. In some cases, these comments also are added electronically to the original assignment; in other cases, each assignment is printed, and comments are marked on paper. Assignments are then returned to the student, either by e-mail or on paper. Compared to mail submission, the reduction on turn-around time results from the higher speed of the network compared with the post office. This decrease in time is extremely beneficial in distance learning environments even though grading time is still the same. Nevertheless, instant feedback is very hard to achieve, especially for large-sized classes.

Automated testing systems eliminate the long delays in turn-around time by providing instant feedback on grading, thus enabling a higher degree of interaction with the user. Some systems also automate the submission process using the network. Usually, tests are limited to multiple-choice or matching questions, since automatic answers in natural language can be extremely complex and usually assume certain restrictions what is being answered(see [9] for a system with some intelligent capabilities). System feedback often varies from a simple grade and a report of the questions missed, to more complex replies in the form of remedial exams and/or mail messages to the instructor (warning about a poor grade from a student, for example).

There are several things to take into consideration when deciding to use automated tests in classes. First, consider the resources involved in administering the test. Every student in the class should have access to a computer with the appropriate testing software installed. A powerful database engine for efficient data collection and storage is a necessity. In the case where answers and scores travel through a network, network reliability and security also is a factor. Second, usability and familiarity with the system are critical, since students should not disperse their attention struggling with bad interfaces when they are supposed to focus on the test's contents. Ideally, both instructors and students should be trained on how to use the system prior to authoring and taking tests. Third, instructors should be aware of the pedagogical limitations of multiple-choice exams. More often than not, most of them completely overlook this problem with the misconception that computer tests implementing multiple-choice exams can substitute entirely for traditional forms of evaluation. Unfortunately this is not the case, and is very important to stress that computer-administered exams are recommended only as a supplement to other types of exams. We will not go in depth into the pedagogical reasons for this statement in this thesis, but instead will recommend possible uses for the system we developed, QUIZIT.

The next chapter provides a literature review of current automated test systems, an analysis of their features and limitations, and the motivations behind the design of QUIZIT.

3. Existing Automated Test Systems

Most of today's automated test systems are part of so called Computer-Based Training (CBT) Authoring packages. CBTs, in turn, are usually part of general-purpose commercial multimedia authoring packages, such as Authorware [14], Toolbook II [1], and many others. These systems provide the ease of presentation software coupled with branching and interactive capabilities, integrating text, sound, graphics, animation, and video with good performance [12]. Nevertheless, the growth of the WWW as a de-facto standard for delivering online courseware [16] combined with the difficulties in integrating the authoring environment for Internet deployment, the complexity and cost of these systems, and the lack of a standard in document format produced by CBTs has been restricting the use of these systems primarily to multimedia presentations in the business world and third-party companies specialized in developing training software. On the other hand, several Web-Based Training (WBT) systems also incorporating test authoring modules have begun to appear [17, 24]. Although these systems tend to be much less powerful than their CBT counterparts, they tend to provide usable interfaces and a more seamless integration of the test modules produced with other WWW-based courseware materials.

The following sections explain in more detail some important features of these test systems, providing a brief classification scheme that will be referred to throughout this paper.

3.1 Authoring Interfaces and Document Representation

One of the problems with most test modules of CBT systems is that each of them has its own authoring environment. Although some systems provide a good variety of features, such as hypertext capabilities, multimedia databases, and digital-media editors integrated into a graphical user interface (GUI), very few allow fully cross-platform development and playback, and only Asymetrix's Toolbook II provides the above features*and* Internet deployment. In addition, the internal document representations of these tests do not comply with any standard and are therefore not easily storable in a format adequate for automatic retrieval purposes. Systems that do provide Internet deployment, on the other hand, assume certain types of browsers and plug-ins in the client side and do not integrate the multimedia content of their tests with WWW servers. Although some of them (like Icon Author and Toolbook II) use a combination of HTML and Java to deliver their packages, only Toolbook II provides some sort of record keeping between the test modules and a "course administration" module over the network. Even though the use of Java enables some interesting real-time interaction capabilities, in simulations for example, the transactional nature of test taking makes a server centered model like CGI (which will be explained in more detail later) more adequate and less resource draining than Java, especially when we take into consideration the importance of legacy systems. In Chapter 6 we discuss future revisions of the system and how Java and remote methods can be used in them.

In spite of being inherently more portable than CBT systems, WBT's authoring interfaces tend to be less sophisticated. Most of these systems directly use HTML or HTML extensions as authoring languages. Unfortunately, there are several problems with this approach. First, HTML is a mostly presentational markup language, and it is constantly

changing. For efficient automatic indexing, better standardization (e.g., pre-defined styles), and higher consistency, it is much better to have structural or descriptive markup [4]. Emphasizing content as opposed to presentation also leads to a more usable and well-defined authoring process, and better document integration with digital libraries. Second, as HTML changes and adds more presentational syntax, documents represented in HTML also tend to change because of new style requirements. As test libraries quickly build up, we may end up with multiple representations of documents with similar content. Thirdly, these systems' languages do not allow *test adaptability.* The next section describes the concept of adaptability in more detail.

3.2 Adaptability or Sequentiality

Normally, the idea we have of quizzes is of one-page paper tests that are marked and then returned with a certain type of performance evaluation (i.e., a grade), and maybe some indication of what we did wrong. This simple process implies only one level of interactivity, since we get a single reply after answering all the questions and submitting the quiz. A more interesting scenario, however, occurs when we start thinking of a quiz as having more than one level of interactivity, where we can get dynamically selected replies as we go along. For example, we could initially answer a set of questions, and depending on our performance, proceed to either a harder or easier set, and so forth. A reply in this case would correspond to a performance report on the question or set of questions answered plus the new quiz to be taken. In other words, we can now think of a quiz as a *sequence* of sub-quizzes, where each consecutive quiz "*adapts*" to the performance on the previous one. These types of quizzes are usually known as *adaptable* or *sequential* quizzes [8]. The pedagogical motivation for adaptable quizzes is the ability to provide a higher degree of interactivity and remedial feedback depending upon prior

performance. For example, if a student does poorly in one quiz, we can provide a second, easier quiz based upon similar content. On the other hand, if a student does extremely well, we can go on issuing harder questions. Actually, we almost could use the metaphor of an oral exam to describe the process, if it weren't the fact that in oral exams, questions can be of any kind, not just simple multiple-choice or matching.

Among the different test systems included in CBTs, only a few provide the ability to author adaptable quizzes. We could not find any WBTs that have this feature. As we will see in more detail in Chapter 4, QUIZIT provides full support for authoring adaptable quizzes at the document level, plus the required run-time system that implements automatic feedback on standard WWW servers.

3.3 Record Keeping

An extremely important feature of test systems is the ability to store log data and results for every quiz taken. These results can then be used by the instructor or evaluator to monitor students' performance on a set of quizzes at any given point in time. In particular, log data can be useful to determine how much time a student spent taking a quiz, the order of quizzes taken (including the number of levels, in the case of adaptable quizzes), and the precise date when each quiz was taken (i.e., begin/end time).

Most commercial CBT systems provide some type of record keeping using client-side databases, whereas others simply record individual scores. On the other hand, very few WBT systems provide extensive record keeping. The great majority only store test scores. When more specific test statistics are computed, it is with the purpose of providing feedback to the student, and not to the instructor. Therefore, these systems tend

to embrace more the philosophy of self-evaluation than of monitored evaluation. In a classroom environment, however, it also is very important that records for all the students in a class can be analyzed together, so that conclusions and comparisons can be made about a class as a whole (e.g., averages, standard deviations, progress reports, and other statistics). Unfortunately, very few of these systems provide centralized database support online, nor do they assure data consistency among the client databases (with the exception of Toolbook II w/ the Librarian module).

3.4 Quantifying performance and scoring methods on adaptable quizzes

Once we have all the performance data and logs, the big question concerns how we should assign absolute scores to an arbitrary adaptable quiz that can be authored with these systems. The answer is that no one really knows. What appears to be a simple problem is actually a very hard one, since different students may be taking different "versions" of the same test. Because no assumptions can be made about the content of each possible sequence of tests taken by a student, it does not make any sense to compare scores of two students who traversed different paths. Several researchers [2, 8] were successful in finding sound scoring methods by making some assumptions about the contents of each sub-quiz, and the elapsed time spent on each one. However, the problem becomes even more complex if we consider the possible network failures that may occur during the process of taking a test across the network, their effect on elapsed time, and the effect on the student's concentration.

Because of all these problems with scoring, the general practice usually is to analyze logs and paths when dealing with adaptable quizzes, placing emphasis on task completion as opposed to score analysis. For example, even if two students go through totally different

sequences, the ultimate goal for both of them should be to complete a pedagogically satisfiable task independently of the sequence traversed. Using this practice, scores would have importance only within a single sub-quiz, to establish a cutoff mark for determining which quiz should be taken next in the sequence.

Most current automated testing systems allow the author to choose what scoring practice, if any, will be used. In most WBT systems, scoring is built into the system, and simple mechanisms like assigning one point for right answers and no points for wrong answers, is often used. Since none of these systems are adaptable, this is not a big problem, as we explained before.

4. The QUIZIT System Design

The next chapters will focus on the design, implementation and use of the QUIZIT system. Section 4.1 shows different scenarios for registering students, authoring quizzes, monitoring students' performance, and taking tests using the system. It also gives an overview of the architecture. Section 4.2 describes the authoring environment and the motivations for using the QUIZIT Markup Language to represent quizzes. Section 4.3 shows the monitor environment, how instructors can keep track of students' records, and the database design. Finally Section 4.4 is a description of the run-time environment.

4.1 Some scenarios and the QUIZIT system architecture

This section describes a typical authoring and monitoring cycle in QUIZIT, through the use of scenarios. Scenarios provide a step-by-step, concrete, yet informal description of possible user interactions with the system, improving our understanding of requirements and, in our case, the instructor's view of the system.

4.1.1 Organization and Setup

Professor Testman gets home after a busy day at the library, gathering references in preparation for his upcoming course on data structures. After logging in to his computer at the university (from home), he gets an electronic copy of the class roster, sent by the department's secretary: "Prof. Testman: CS3604 : 300 students / undergraduate level" -- can be read in the top header. Still in the same machine, he logs into the QUIZIT account and starts up a QUIZIT registration session. The system promptly asks for his ID and password, course name, and then roster file name for the class. After typing in the information, the system automatically generates initial logins and passwords for each of

the 300 students in the class. If e-mails are available in the roster, it also sends messages to each of the students in class informing them that their accounts have just been set up. This information includes a registration page URL they should refer to in order to change the temporary password assigned to them by the system, along with their login ID and social security number. The whole registration process takes about five minutes. After some thinking, Dr. Testman decides that his course will be organized in five different units, with three quizzes per unit. Quizzes in different units can be taken in any order, whereas quizzes within a unit may only be taken sequentially . To setup this structure, he executes a "unit setup" program in the QUIZIT account, which generates the database tables that keep track of which quiz each student is taking within that unit.
The registration and unit set-up process are illustrated in Figures 2a and 2b.

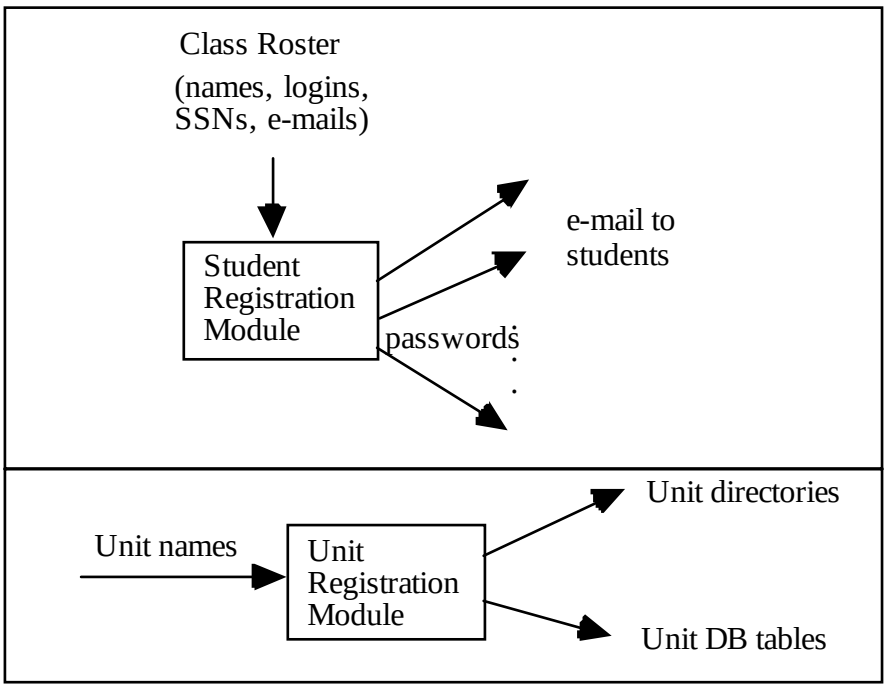

Figure 2a, 2b: User registration and unit setup in QUIZIT environment

4.1.2 Authoring quizzes

Dr. Testman then starts preparing the quizzes for his "linked list" unit. He first goes to the QUIZIT authoring tutorial site [20] and grabs a copy of the QUIZIT Markup Language (QML) template, which contains samples of different types of questions and a suggested outline for a general quiz. He could alternatively retrieve some of his old quizzes available in the quiz bank and start from there. Using any text processor in his machine at home, he writes the first three "linked list" quizzes using the predefined, content-oriented elements in the markup. After preparing the documents, Dr. Testman runs the QML compiler, which translates each document into one HTML page and one answer file. After fixing some compilation errors, he activates his browser and double-checks the final presentation form of his quizzes. Dr. Testman is now ready to upload the files to the specific unit directories in QUIZIT. The authoring and publishing process is complete as soon as the files are placed in the right directories. Figure 3 illustrates the actions performed by Dr. Testman:

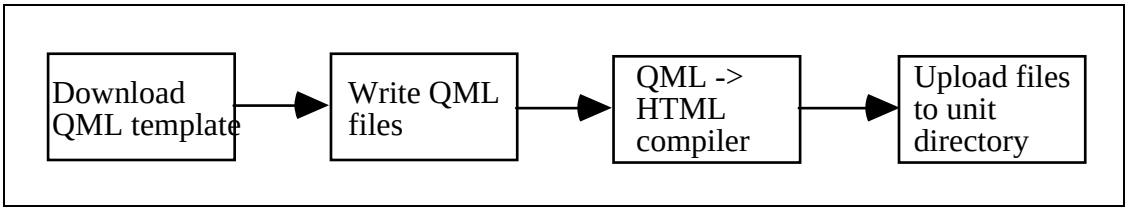

Figure 3: Functional view of the authoring process in QUIZIT

4.1.3 Monitoring Students

Once students start taking quizzes, it is time to see how well they are doing. In particular, Dr. Testman is especially interested to see how students are progressing through the

material and to identify students who may be lagging behind. Using his browser, he accesses the monitor page (which is password protected) and from there, “activity logs”. On that page he has the option of accessing all or specific units. He chooses “all units”, where he sees the activity report of every student. He notices that, although he had already published units 6, 7 and 8, no student has completed unit 4 yet, and some of them are still in unit 2. He takes note of these students’ names. Later on, he will send messages to these students asking if there are any problems. In addition, he will again check these students’ answers to the quizzes on unit 2, to see if they seem to be stuck on a particular quiz.

4.1.4 Taking tests

After Dr. Testman announces to class the posting of a new quiz, Mr. Testaker, a student in his class, decides to go home, quickly go over the material, and take the open-book, open-notes quiz. He does that by running his WWW browser, pointed to the QUIZIT URL for that course. On that page, he has the options of reviewing old quizzes or taking a new quiz. After entering the “new quiz” page, entering his login ID and password, and selecting the appropriate unit, Testtaker sees the first quiz screen, which contains the start time for that quiz and the quiz itself. When he is ready, it is time to submit the quiz by pressing the submit button at the bottom of the quiz page. In 2 seconds the system replies with his grade, information about the next quiz to be taken, a report of the questions missed and the answers, hyperlinked. Following the hyperlinks will take Testtaker to the quiz page again, where he sees the questions missed highlighted. After a quick review of his mistakes, Testaker decides to break and take the rest of the quiz later which he may do by simply leaving his browser at this point. When he comes back later, he will be able

to come back to the exact same point and take the next quiz. Figure 4 below represents the process of taking a quiz, from the student's point of view.

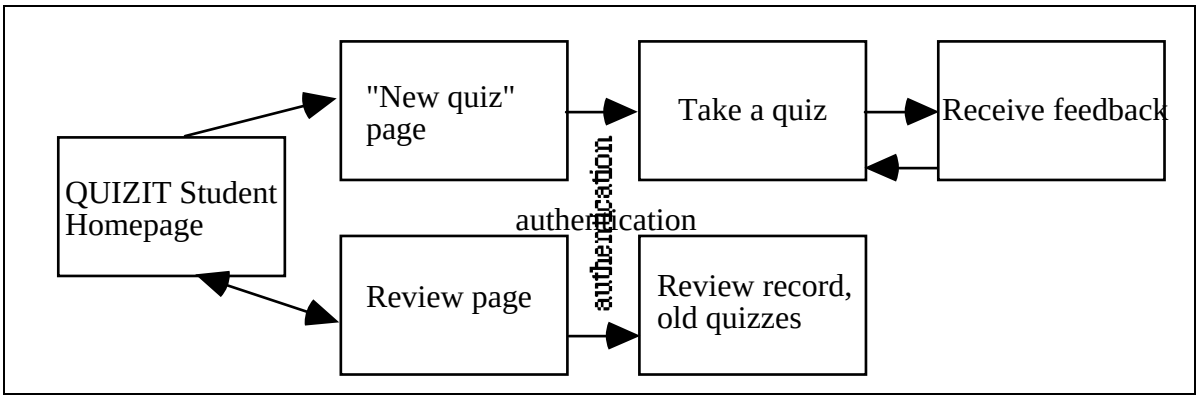

Figure 4: Taking a quiz: *note that at any point a student can leave the system and come back later*

4.1.5 Architecture Overview

Figure 5 represents an architectural overview of the different modules of the QUIZIT system. From the scenarios above, you should already have an idea of the functionality of each one of the modules in the system. A detailed and more technical description of each module and the interactions among them is the subject of Sections 4.2, 4.3, and 4.4.

The major difference between QUIZIT and other WBT systems is QUIZIT's heavy communication between the WWW-server and the database server. This communication is achieved by using the Common Gateway Interface (CGI), a mechanism provided by the WWW server that allows transactions to occur between a WWW browser and external processes running on a server machine. Most WBT systems have a very limited interaction model (remember that none of them provide adaptive tests) and therefore do not rely heavily on advanced CGI techniques like state recording and the use of server-

side cookies. In Sections 4.3 and 4.4, where we talk about QUIZIT's run-time and monitor modules, we will provide some background information about CGI and database gateways. It is not necessary now to understand how CGI interconnects the different modules in the diagram.

One important aspect to note is the decoupling between the authoring modules and the run-time engine. In other words, instructors do not need to access the database or WWW server during the authoring process. In fact, the only requirement is to have some kind of word processor software working on a local machine and the ability to transfer the quiz document electronically, using either a diskette or a network. Further, the compilation process can also be done in the client, since the compiler itself is portable among the different popular platforms (Mac, PC, UNIX). The files generated can be placed directly in the appropriate directories in the QUIZIT account on the server machine.

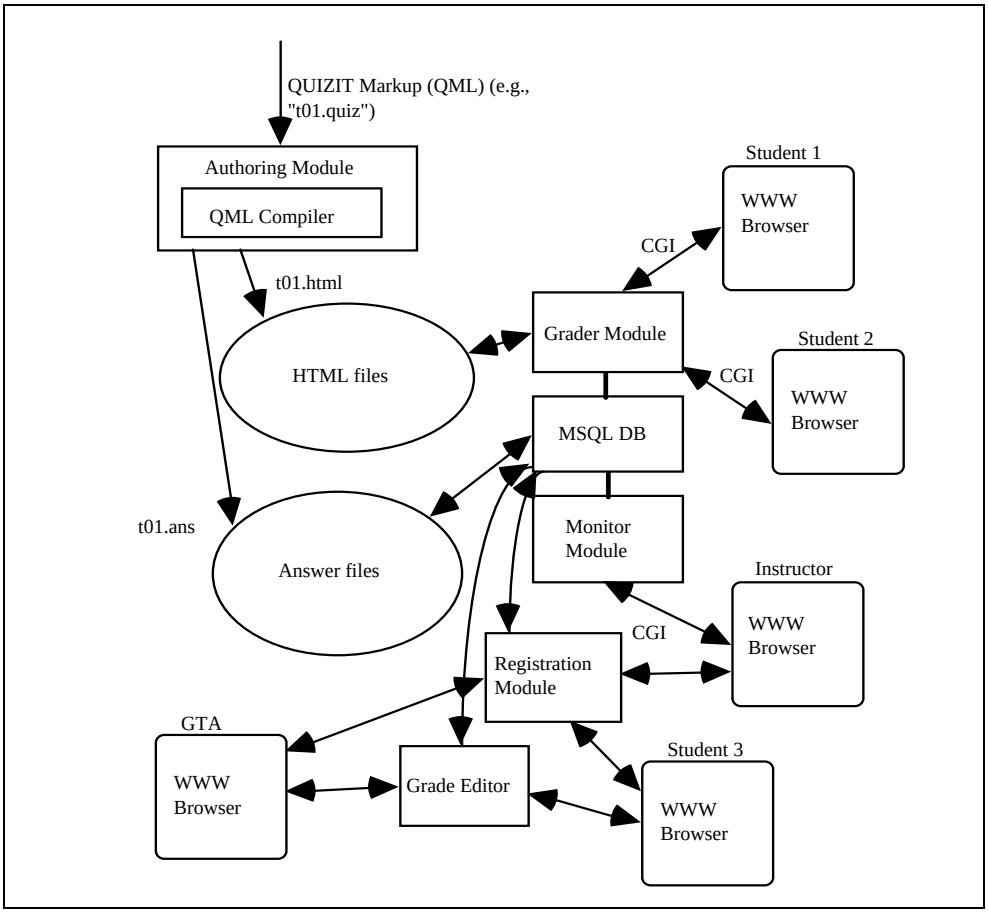

Figure 5. The QUIZIT System Architecture

It also is possible to have the database server running on a separate machine, since the Application Programming Interface (API) calls to the database server we are using can be accessed remotely. However this is not recommended, since the WWW-server and the database server have a very tight coupling. This issue will be explored later in Section 4.4.

4.2. The Authoring Environment

Since our primary goal is to make the process of authoring electronic quizzes both *portable* and *reusable*, one of the central design issues of the authoring environment in

QUIZIT is the use of standards to represent documents used by the system. In addition, standardization gives us the ability to retrieve electronic documents more efficiently using automated techniques, which becomes a necessity as potentially large sized test banks build up. In particular, since the WWW is our target delivery medium, the publishing process tends to be less and less centralized, increasing even more the need for consistency and content-oriented representation in documents.

One of the major problems with other training systems is the lack of standardization. In spite of many efforts from standards organizations, there are many aspects of software, hardware, and procedures that are inconsistent and incompatible among these systems. Sometimes this is intentional, since certain vendors hope to trap a segment of a user community so as to assure product fidelity and future sales. CBT systems, for example, all tend to represent test documents in a different way. Nevertheless, systems that provide WWW authoring have the option of exporting an HTML representation of the test that can be uploaded to the desired location. WBT systems, on the other hand, use HTML or HTML *extensions* directly as authoring languages. In the next subsection, we will describe the use of markup languages during the authoring process. Further, we analyze the advantages and disadvantages of using a specific type of markup language, HTML. Finally, we compare HTML to QML and our approach in trying to create a document type definition (DTD) for a markup language which heavily emphasizes descriptive or structural content.

4.2.1 Markup languages on WBTs and the need for content-oriented syntax

As we mentioned before, one of the major advantages of having a standard representation for our quiz documents is the ability to integrate and process documents more easily.

Nevertheless, nothing was said about what kind of standard we should use. This subsection goes over the different alternatives, and explains the motivation for using QML in our project.

The obvious choice, one might first think, is simply to use HTML, since it is a popular markup standard (although every browser vendor have been trying to push its own version) and is used by all the WWW-browsers available today. This is the approach used by some WBT systems and the intermediate representation that CBT systems use for WWW publishing.

Unfortunately, major problems arise because of some presentational and contextual aspects of a quiz. First, a quiz requires very specialized HTML syntax, including specific form features (you have to be able to enter your answers in a page) that are unfamiliar to most authors. In particular, since there are only limited types of questions that can be used on a multiple-choice based quiz (e.g., true-false, multiple-choice, matching, etc.), the process is much less time consuming if an author does not have to worry about the type or layout of every question on the quiz. Second, it is not trivial to map the structure of a quiz document to the HTML elements used to represent it. For example, one question on a quiz can have two different representations in HTML. Figure 6a and 6b illustrate the point with examples of two different formats representing a simple true-false question using HTML.

```
<HR>
<B>Question 1</B><BR>
By default, almost all WYSIWYG word processor programs use <B>proportional</B> spaced
fonts (type faces) for normal paragraph display.<BR>
<INPUT TYPE="radio" NAME="q01" VALUE="TRUE"><B>TRUE</B>
<INPUT TYPE="radio" NAME="q01" VALUE="FALSE"><B>FALSE</B>
<P>
<HR>
```

Figure 6a: Example of a true-false question using HTML. ***This particular question was taken from the second quiz of CS1604, taught by Dwight Barnette.***

```
<I>Question 1</I><P>
By default, almost all WYSIWYG word processor programs use
<U>proportional</U>
spaced fonts (type faces) for normal paragraph display.<P>
Type either TRUE or FALSE in the text area below:
<INPUT TYPE="textarea" WIDTH=5 MAXWIDTH=5><P>
<HR>
```

Figure 6b: Same question with presentational markup included

Although it seems at first that presentational standardization in HTML can limit creativity and flexibility, studies [5, 8] report that in computer-based testing systems more homogeneous and consistent interfaces lead to better concentration and performance from students.

A third, more technical reason for not using HTML is the heavy use of server-side *CGI cookies* in the communication between the HTML page and the run-time grader system. As we will see later, the use of server-side cookies and CGI cause the final HTML document to be much more complex (e.g., use of hidden fields, script references). By hiding some of that complexity and showing instructors only the new syntax, we can make our documents much more readable and usable.

4.2.2 The Standard Generalized Markup Language and the QUIZIT Markup Language (QML)

The markup language used to represent electronic quizzes in QUIZIT was defined using the Standard Generalized Markup Language (SGML, ISO/IEC 8879) [3, 7]. SGML is a meta-language that enables the definition of general markup languages using *document type definitions*, or DTDs. A DTD can be thought of as an attribute grammar, where each left-side production defines an *element* . In the document itself, an element can be referred to by using its generic identifier (GI, i.e., the tag used to represent the element, such as <TITLE> to represent the document title, for example) and may include attributes as well.

Figure 7 shows the QML DTD. If you have seen grammar definitions before, you should be able to understand quiz syntax from the DTD. In the next subsection, we will explain the elements in more detail. For a complete authoring guide, please refer to the QUIZIT authoring tutorial available on the WWW [21].

Although HTML also is defined formally using SGML, it is important to emphasize once again the differences between HTML and QML. First, HTML is a general language for all hypertext documents rendered by a WWW-browser. It includes both content and

presentational markup elements for documents with general structure (i.e., from personal home pages to complex tutorials). QML, on the other hand, is specifically a *quiz markup language*. In such a restricted domain, we can achieve a high presentational homogeneity and specialization. Second, as HTML evolves because of market requirements of browser vendors, new presentational elements are being thrown into the language. In HTML 3.0, for example, we even have a <FONT> element with color and size attributes! QML does not define any presentational elements, although it provides the presentational and hypertext flexibility of HTML by enabling HTML markup to be embedded within textual QML elements only. For example, you can include figures, movies, or animations inside a question instruction element (<INSTRUCTION>) :

<INSTRUCTION>
<P>After watching the <A HREF="movie.mov">following movie</A> and looking at the <A IMG SRC="picture.gif">picture</A>, answer the following question.
</INSTRUCTION>

```
<!--  SGML DTD for Simple QUIZ object         -->
<!--  By Lucio Cunha Tinoco                   -->
<!--  DATE: 9/21/1996                        -->

<!--
<!ENTITY % quiz PUBLIC "-//VT//QUIZIT System 1.0//EN">
-->

<!-- RCS INFO: $Id: quiz.dtd, v 1.0 1996/05/09 13:40 ltinoco Exp $ -->

<!-- Defines the different types of question -->
<!ENTITY     %     qtype  "mchoice| matching| truefalse | fillin">

<!-- Special Characters -->
<!ENTITY     lt     CDATA  "&#60;" >
<!ENTITY     gt     CDATA  "&#62;" >
<!ENTITY     quot    CDATA  "&#34;" >
<!ENTITY     amp     CDATA  "&#38;" >

<!ELEMENT     quiz     - - (title?,instruction?,(%qtype;)+, reply?)>
<!ATTLIST     quiz
     id            ID             #REQUIRED
>
```

```
<!-- Title is simply the quiz's title -->
<!ELEMENT     title               - - (#PCDATA)>

<!-- General instructions for the quiz -->
<!ELEMENT     instruction          - -  (#PCDATA)>


<!-- A paragraph is required within descriptions, and answers. Inside a
paragraph,
any HTML markup is valid, including hyperlinks -->
<!ELEMENT     p               - -  (#PCDATA)>

<!-- An item is an element used by a matching question. Its attribute specifies
the match. When defining an answer for a matching question, the attribute
"value" will hold the value to be matched against. Please see the examples. -->
<!ELEMENT     item          - -     (#PCDATA)>
<!ATTLIST     item
     matches          NAME          #REQUIRED
>

<!-- The question's description can be either a sequence of paragraphs -->
<!-- or an itemized list, in the case of matching type of questions -->
<!ELEMENT     description     - -  (#PCDATA)     >


<!-- Each question, be it truefalse, mchoice, or matching, has a description
followed by one or more possible alternatives. The attribute "id" is used when
this question is referred to by some anchor, like a NAME tag in HTML -->
<!ELEMENT     (%qtype;)     - - (description, item*,  answer+, hint*)     >
<!ATTLIST     (%qtype;)
     id          ID          #REQUIRED
>


<!-- A hint is a list of paragraphs that goes after each set of answers for a
question -->
<!ELEMENT     hint          - - (#PCDATA)>

<!-- Answer is defined to be the same for the different type of questions, but
its -->
<!-- semantic meaning varies for each question type. Please take a look at the
examples -->
<!ELEMENT     answer          - -  (p)>
<!ATTLIST     answer
     value   CDATA  #IMPLIED
     points  CDATA  #IMPLIED
     id             ID     #REQUIRED
     label          CDATA  #IMPLIED
>

<!-- A reply action is associated with one question-group. It specifies actions
to be taken in case the student doesn't obtain the minimum number of points
specified
in "cutoff". -->
<!ELEMENT    reply  - -    EMPTY>
<!ATTLIST    reply
       remedial     CDATA  #IMPLIED
       next         CDATA  #IMPLIED
       cutoff       CDATA  #IMPLIED
>

<!-- END OF QUIZ DTD  -->
```

Figure 7: QML DTD

4.2.3 More on QML

The following example, taken from the CS5604, Information Storage and Retrieval courseware, illustrates the markup of a quiz document on inverted files and boolean systems.

Each quiz document starts with a top-level element, QUIZ, which delimits the quiz contents. A quiz also has a required ID attribute, specifying a path to the HTML file generated by the QML compiler. Each quiz document generates exactly one HTML file, plus an answer database file. Figures 10a and 10b show the final HTML document as viewed by a WWW browser.

```
<QUIZ CGIBINDIR="/quizit-cgi-bin" ID="IFquiz/IFA3.html">
<!-- c 1996 E. A. Fox & VPI&SU. Reproduced with permission. -->
<TITLE>IF Unit Quiz - A3</TITLE>
<INSTRUCTION><p>This is the quiz you should continue with, as individual
work according to the honor code, for credit on the IF unit.
Unless stated otherwise, each correct answer is worth 2 points.
If you master this, you will proceed to the 2nd of two parts
of the IF unit quiz.  Otherwise you will be shown where you
did not give correct answers. When you understand those answers,
and after studying to be sure you fully comprehend this part of
the unit, you can take an oral quiz with the
instructor.
<p>When you have successfully completed one of the A versions, you then
can proceed to successfully complete one of the B versions.</INSTRUCTION>

<TRUEFALSE ID="q0001">
<DESCRIPTION>
True False Question:
</DESCRIPTION>
<ANSWER POINTS="2" VALUE="true"><P>
An inverted file is found in most
commercial library retrieval systems.
</P></ANSWER>
</TRUEFALSE>

<MCHOICE ID="q0002">
<DESCRIPTION>
Which of the following structures used in an inverted
file is simplest to implement and requires the least
amount of space?
</DESCRIPTION>
<ANSWER VALUE="right" POINTS="2" ID="my1" ><P>
sorted array
</P></ANSWER>
<ANSWER VALUE="wrong" POINTS="2" ID="my2" ><P>
B-tree
```

```
</P></ANSWER>
<ANSWER VALUE="wrong" POINTS="2" ID="my3" ><P>
trie
</P></ANSWER>
<HINT> Select the single best answer.
</HINT>
</MCHOICE>

<MCHOICE ID="q0003">
<DESCRIPTION>
The textbook discussion of inversion speeds
indicated which of the following to be fastest?
</DESCRIPTION>
<ANSWER VALUE="wrong" POINTS="0" ID="my1" ><P>SIRE</P></ANSWER>
<ANSWER VALUE="wrong" POINTS="0" ID="my2" ><P>SMART</P></ANSWER>
<ANSWER VALUE="right" POINTS="2" ID="my3" ><P>FAST-INV</P></ANSWER>
<HINT> Select the single best answer.</HINT>
</MCHOICE>

<MATCHING ID="q0004">
<DESCRIPTION>Match the following
Boolean expressions
with
English statements.
</DESCRIPTION>
<ITEM MATCHES="q00c"><P> Documents about information but
not also about retrieval </P></ITEM>
<ITEM MATCHES="q00a"><P> Documents about both information, retrieval </P></ITEM>
<ITEM MATCHES="q00b"><P> Documents about either of information,
retrieval </P></ITEM>
<ANSWER LABEL="q00a" POINTS="2"><P> information AND retrieval
</P></ANSWER>
<ANSWER LABEL="q00b" POINTS="2"><P> information OR retrieval
</P></ANSWER>
<ANSWER LABEL="q00c" POINTS="2"><P> information NOT retrieval
</P></ANSWER>
</MATCHING>
<REPLY REMEDIAL="done" NEXT="IFquiz/IFB1.html" CUTOFF=8>
</QUIZ>
```

Figure 8: QML for CS5604 quiz.

Currently, QUIZIT supports four types of questions: multiple-choice, true-false, matching (all shown in the example above) and fill-in. Multiple-choice questions, represented by the `<MCHOICE>` generic identifier (GI), enable multiple selection of answers (i.e., one or more correct answers). True-false questions (`<TRUEFALSE>` GI) have a descriptive statement, followed by a mutually exclusive, boolean choice (i.e., true or false). Matching questions allow a one-to-one or many-to-one matching between a series of items and a series of answers. Fill-in questions (`<FILLIN>` GI) allow simple, case insensitive string matching.

```
<FILLIN ID="question5">
<DESCRIPTION><P>What is the name of the inverted file construction
algorithm, developed by Edward Fox and others at Virginia Tech, that
inverts the file in different memory loads, merging the results
later?</P>
</DESCRIPTION>
<ANSWER VALUE="FAST-INV" POINTS=10 ID="my1"><P>Put answer
here</P></ANSWER>
</FILLIN>
```

Within each question element, one or more answers can be defined by using the ANSWER element (<ANSWER> GI). This element contains an optional label (required for matching questions) that uniquely identifies that answer, the number of points assigned to the question (negative points also can be assigned), and a textual description of the answer.

As mentioned before, HTML elements are allowed within any textual QML element (e.g., `description`, `instruction`, `title`, `answer`, or `hint` GIs). This allows multimedia and hypertext content to be included as part of a quiz. In fact, entire tutorials can be modeled using QML because of this feature. Note, however, that the ability to embed HTML notation within QML elements does not reduce the descriptive power of the document, since each HTML element is "wrapped" by a QML element with descriptive meaning.

Adaptability is put into the quiz structure by using a "REPLY" GI. The two attributes of REPLY specify what quizzes should be taken depending on the student's score. The following syntax, taken from the "IFquiz/IFA2.html", CS5604 quiz, means that a student who achieves 8 or more points will take quiz "IFquiz/IFB1.html" next. Otherwise, if less than 8 points are achieved, this student will have to take quiz "IFquiz/IFA3.html"

```
<REPLY REMEDIAL="IFquiz/IFA3.html" NEXT="IFquiz/IFB1.html" CUTOFF=8>
```

Figure 9: Use of the REPLY GI in IFA2 quiz.

Although very simple to use, this syntax is very powerful. By combining question grouping on a document and reply elements appropriately, one can author quizzes like the computerized, adaptive GRE [5] , for example (where you have one question and one reply element per page), or just simple, one-page quizzes where no remedial or progressive actions are required. In general, you can think of an entire quiz as a binary directed graph data structure, where each node is a quiz document and the one or two edges leaving it correspond to the two possible actions taken by the system ("remedial" or "next"), according to the branching value determined by the CUTOFF attribute.

Figure 10a: QUIZ header and multiple choice question

Figure 10b: Document Source in HTML

4.2.4 The QUIZIT Compiler

The QML to HTML translation is done by a special QML compiler. Several design goals had to be taken into consideration when choosing what type of compiler should be designed and how to implement it. This Section presents a list of those goals, and a detailed explanation on how we implemented them:

• ***Portability*** : source code portability was a primary concern during the design process. In general, one should be able to author quizzes using just a text processor (e.g., plain unformatted text), and compile it on any client machine. This is consistent with the markup philosophy of SGML and clearly stated in the Association of American Publishers (AAP) standard:

> to provide a logical way to represent special characters, symbols, and tabular material, using only the ASCII [plain text] character set. By following these procedures, any manuscript can be processed on any computer [23].

• ***Usability:*** The compiler should be small, simple and specialized, with transparent operation for the user and clarifying error message output. Further, it should be easy to distribute and install on every machine.

• ***Reusability:*** Changes in the QML DTD or in the presentation requirements should only cause minimal changes in the compiler. In particular, both the DTD and the HTML code generator should be easy enough to read as to maximize the reuse and extensibility for changes.

Three implementation options were considered during the analysis of these fundamental requirements:

• ***Build a compiler by hand***, without the use of any generation tools or validating parsers: Since the attribute grammar used in our DTD can be reduced to an LL(1) form, it would be simple enough to write a predictive parser for QML. This would provide maximum portability, but at the cost of lower reuse, since the grammar specification would have to be hard-coded in the parser. Consequently, changes in the DTD would cause direct changes to the recursive descent parser in C code.

• ***Use a validating SGML parser*** and build the other parts by hand. This approach has the advantage of providing direct DTD processing and complete SGML compliance. However, even for a simple grammar like QML, this would require the use of existing validating parsers that are not portable across the most commonly used platforms in university environments (e.g., Macintoshes), and unnecessary complexity in modularization and packaging. Changes in the DTD would still require direct changes in the parser's code, since existing SGML parsers do not allow the definition of action rules for grammar productions (in other words, they just *validate* input).

• ***Use of Lex and Yacc*** [11] for lexical analyser and parser generation, plus a handwritten code generator in C: This was the approach chosen for several reasons. First, translating a DTD grammar to Lex and Yacc specifications is relatively straight-forward. Because Yacc generates parsers for more general LR(1) grammars, only minimal grammar re-engineering is needed to transform the attribute grammar specification on the DTD to an LR(1) grammar. Second, once the token and grammar specifications are written, further

updates only require simple changes in the specification. Third, better error reports can be given. Because the compiler is tailored to QML, more specific and clarifying error messages can be produced. Fourth, the code generated by Lex and Yacc is very portable. As we are in the current process of adapting the code done in UNIX platforms to Macintoshes, we are finding very little problems with the ANSI C code generated by these tools. As a consequence we are able to concentrate almost all of the development effort in designing the graphical user interface for the compiler.

Changes in the HTML presentation of the document, however, still require changes in C source for the code generator, which is implemented without any tools. Nevertheless, the simplicity of the design combined with the power of the first-pass parser generated by Yacc, enabled us to write a very concise and simple recursive descent code generator.

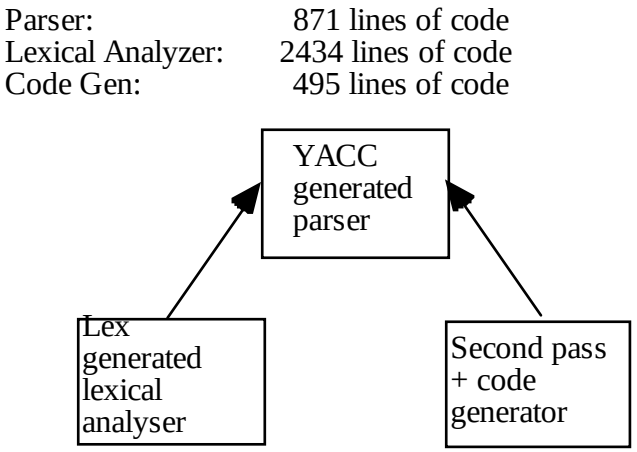

Figure 11a: The structure of the QUIZIT compiler

Figure 11b shows the tail end of the output generated by the QUIZIT QML compiler during a typical session using the UNIX platform.

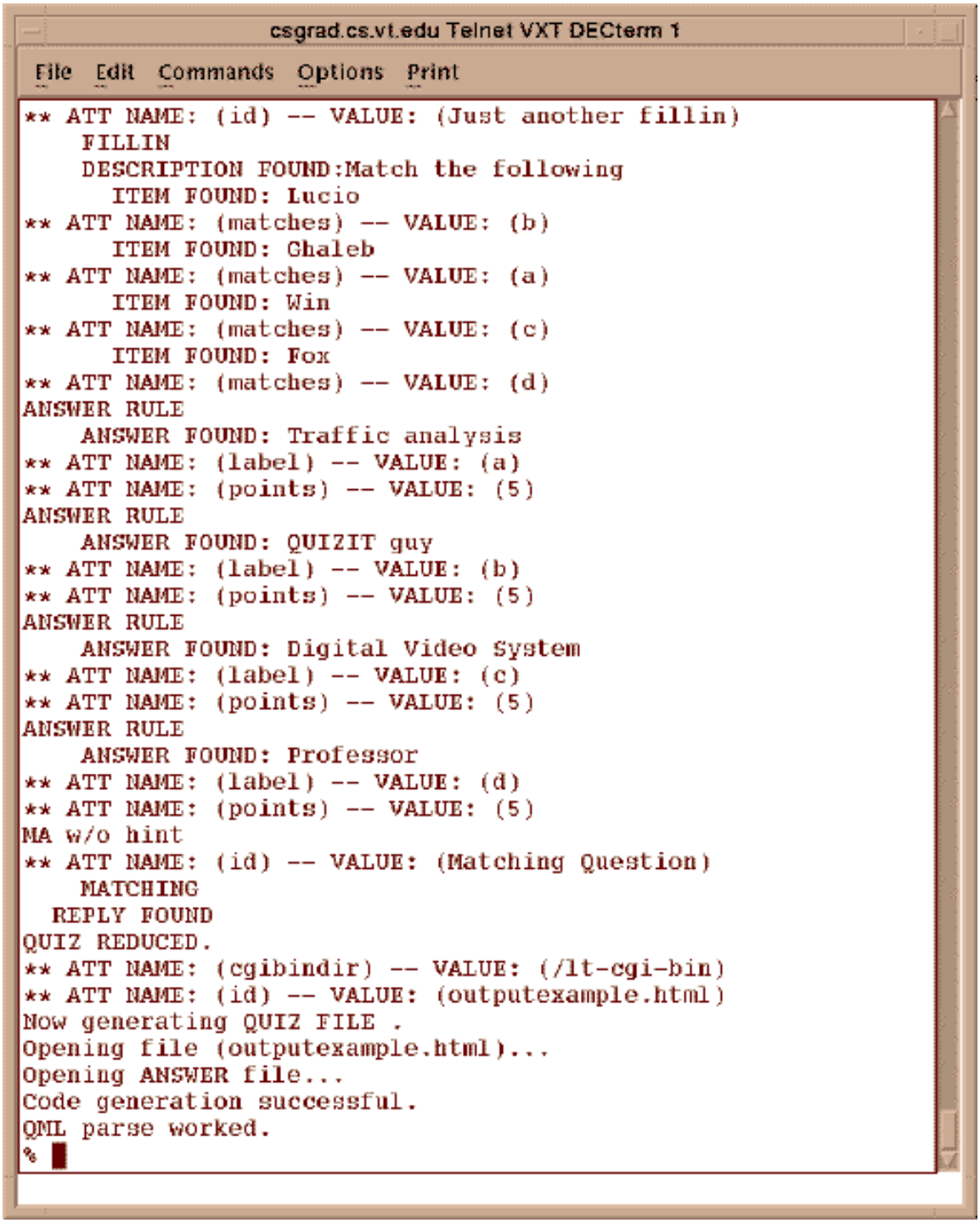

Figure 11b: QUIZIT -- compilation session

4.3 The Monitor Environment

One major problem instructors face when deciding to apply self-paced instruction in today's universities is the difficulty in organizing the huge amount of asynchronous class-related data generated by students, possibly from different locations. As class enrollment in universities keeps growing, instructors quickly become victims of information overload. It is clear that efficient record keeping, especially for large sized classes, is becoming a necessity.

Although the automation of grading electronic quizzes can significantly reduce some of this data overload and processing delay, grading results and performance reports still should be carefully analyzed by instructors for each and every student in the class. In fact, electronic tests are only valuable if a comparative interpretation of results can be done in an efficient manner. This is especially true when using adaptable tests, since objective scoring methods may not be applicable.

In addition to individual performance tracking, there also is a need to develop tools that can help in class administration, which in turn involves registering students into the system, generating progress reports for the entire class, calculating test averages, etc. One technical implication of this requirement is the importance of insuring data consistency and an up-to-date representation of a class database at any point in time.

Currently, the majority of CBT and WBT systems do not provide any means of database management and administration over the Internet. In most CBT systems performance data is collected in the client only, with a few systems providing the option of exporting a copy of the database contents into a text file. Only one system surveyed [14] enables

central database record keeping. The WBT systems only provide minimum database support, primarily using a flat text file representation of scores and answers for individual tests. None of the systems provide group account registration, networked grade editors and real-time performance monitoring.

The *QUIZIT monitor environment* is the set of WWW tools designed to accomplish the following requirements:

• Automate student registration into the QUIZIT system, using an electronic copy of the class roster. This is called the **student registration module** (contribution of Tommy Johnson). Further, keep an up-to-date list of names, passwords, and student IDs that can be easily accessed at any point in time.

• Provide a log of all the activity taken place in the system, including login/logout times on an individual or class basis, IDs, and scores on every quiz. This is called the **activity log module.**

• Provide a snap shot of the progress of every student in a class, including number of units completed, quizzes and cumulative scores for each unit, and an individual answer report for every quiz. This is called the **class roster module.**

• Allow authorized users (e.g., instructors or GTAs) to adjust scores, add or delete users at any point during the course. In addition, enable change or update of passwords and permissions. This is the **maintenance module**.

Each one of these modules consists of a series of WWW pages and server-side *Common Gateway Interface* (CGI) programs. QUIZIT's CGI programs provide the communication between our WWW-based interface and the underlying relational database. In general, CGI gives programmers a way for HTML pages to call external programs and get back the results. The entire process is a communication protocol between the WWW browser, the WWW server, and the external program. The following example list summarizes the actions in this protocol:

1. A user calls a CGI program by clicking on a link or by pushing a submit button on a WWW page.

2. The WWW browser contacts the WWW server asking for permission to run the CGI program.

3. The WWW server checks the configuration and access files to make sure the requester is allowed access to the CGI program.

4. The WWW server checks to make sure the CGI program exists.

5. If it exists the CGI program is executed.

6. Any output produced by the CGI program to *stdout* is redirected to the WWW browser.

7. The WWW browser displays the CGI output.

Information can be passed from the WWW browser to the CGI program in a variety of ways, and the program can return the results with embedded HTML tags, as plain text or as an image. The WWW browser interprets the returned results just like any other document. This process provides a powerful tool for running virtually any program, and allows developers to access any external database that provides an application programmer's interface (API). Another advantage is that instead of having to design a front-end application that is tied to a specific database, the front end can remain the same no matter what type of database is used underneath.

Before explaining the functionality of each module in the monitor interface of QUIZIT, we should go over the data model used by the system. The next subsection presents a description of the database tables used by the system.

4.3.1 QUIZIT's Database Design

QUIZIT maintains two databases for each class account, the *authentication database* and the *activity database*. Having two databases allows us to implement different security levels for each of them. For example, one can limit access to the authentication database only to the instructor, but grant access to both the instructor and the GTA in the activity database. Tables 1a and 1b show the relational structure defined in the authentication database.

Table 1a: Authentication table

Field Name and type	***Purpose***
UID: integer (not null)	Primary key. Usually a social security number
FNAME: 15-character string (not null)	User's first name
LNAME: 30-character string (not null)	User's last name
EMAIL: 30-character string	User's e-mail address
PASSWD: 60-character string	Encrypted password
PERM: integer (not null)	Access permissions

Table 1b: QUIZIT's permission schema used by the registration module

User class	***Instructor***	***GTA***	***Others***
Types of access	R/E/D	R/E/D	R/E/D
Binary=Octal code (example)	111=7	110=6	000=0

Passwords in the authentication database are encrypted using the UNIX encryption mechanism (the same used to encrypt system passwords). This is accomplished by the use of the *crypt* function, available in the UNIX system library. A UID or *cookie*, as will be explained later, is a session token used by the run-time engine to keep track of the identity of a quiz taker during a session. The e-mail field is used by the registration system to send temporary passwords to students and to communicate system messages when needed. The permissions field is a three-digit octal code that defines the user's

access rights for account registration and maintenance. This code is similar to UNIX's files permission schema. Each digit represents permissions to *read* (R), *edit* (E) or *delete*(D) the QUIZIT account for that user. A user with code 760, for example (shown below), can have its account read, edited and deleted by the Instructor (represented by the first octal digit); read and edited by the GTA (second digit), but no other user (third digit) can either read, edit or change any of its records.

Table 2 below shows three possible records in an authentication database. We implemented first and last name as two different fields so the database engine can sort by last names more efficiently.

Table 2: Possible records in tables of the authentication database.

UID	***First_name***	***Last_name***	***Email***	***Password***	***Permission***
246754669	Lucio	Tinoco	tinoco@vt.edu	$tuthvvb1*	777
282826155	Joe	Doe	joedoe@vt.edu	###fjh4u47	611
115242423	Frank	Capra	capra@hw.com	475646gd%	611

The activity database holds all the state information of QUIZIT, including answers, scores, submission times, and the progress of each student in the class. The following tables show the specifications for the database.

Table 3: Unit table in activity database, with one record per unit.

Field name and type	***Description***
UNITID (integer)	Primary key. Uniquely identifies a course *unit* .
UNAME (15-character string)	Unit's name

Table 4: Quiz table, with one record per quiz in the system

Field name and type	***Description***
QID	Quiz ID. Identify each quiz in the system
QNAME	Quiz name. Usually a system file name. (e.g. "/u5/tinoco/quiz1.html")

Table 5: Student_log table, with one record per student in class

Field name and type	***Description***
UID (User ID, integer)	Student's ID
UNITID (integer)	Unit's ID
CUMSCORE	Cumulative score for that unit
QUIZ (integer)	Last quiz taken. Represented by a quiz ID (QID)
NEXT (integer)	Next quiz to be taken. Also a quiz ID (QID)

Table 6: Quiz_log table, with one record per quiz taken

Field name and type	***Description***
UID (integer)	User ID
QID (integer)	Quiz ID. The quiz just submitted
SCORE (integer)	Score for that quiz
UID (integer)	Current unit
BEGIN (integer)	Begin time, using UNIX's system time
END (integer)	End time, or "PROGRESS" flag, in case

Table 7: Answer_log table, with record for each question answer by a student.

Field name and type	***Description***
UID (integer)	User ID
QID (integer)	Quiz ID
UNITID (integer)	Unit ID
QTID (integer)	Question ID
QANS (30-character string)	User's answer to the question
QTYPE (2-character string)	Question type: can be either MC (multiple choice), TF (true-false), MA (matching) or FI (fill-in)

QUIZIT assumes each course is represented by one or more independent *units*. Quizzes in different units may be taken in any arbitrary order, and cumulative scores are computed on a per unit basis. Each unit, in turn, can have several quizzes. These quizzes, however,

must be taken according to the sequential thread defined by their adaptable structure. Scores, begin/end times, and answers for each question are recorded for each quiz taken, and for each student taking it.

Monitor modules in QUIZIT are simply HTML-based interfaces and CGI programs for selecting, updating, inserting or deleting records from the tables described above. Although one could directly use a query language to do the same, having a structured WWW interface with predefined queries is a more usable approach. In addition, authorized users monitoring the system are not required to remote login into the system for submitting queries.

The first monitor mode is the *registration* module. This module enables a distributed method of student registration in the system. Section 4.1.1 presented the scenario for a typical registration process. Figure 12 further illustrates this process by showing screen shots of the QUIZIT registration interface.

Figure 12: QUIZIT page in CS1604, with a description of the registration process.

The following list summarizes the process, from a developer's perspective:

1. Using the first and last name, social security number, and e-mail data from the password requester page, a CGI script accesses the authentication database to see whether the student is already registered in the system.

2. If it is a new student, a new password is generated by arbitrarily picking up strings from a compressed dictionary file. In addition, the name information is validated against a class roster file pre-loaded into the system. If the information is valid, the student information also is inserted into the database.

3. If, on the other hand, the student already is registered in the system, a new password is generated using the same method.

4. A confirmation stub, including the student's login and password information is then e-mailed back to the student's address.

5. The student can use the assigned login and password information to take quizzes, using the login page.

There are two security concerns to be discussed. First, the registration and login process transmits unsecured (i.e., unencrypted) passwords, either using e-mail (registration) or the browser (login). Consequently, one can potentially tap the network and get access to the login/password information. Second, if a browser is being shared by more than one person, one could in theory get access to the client's cache and read the password stored there. Currently, there is only one plausible solution to these two problems: client-side encryption. Nevertheless, this would impose severe restrictions on the system's overall portability, since secure servers today are expensive and mostly proprietary. As a consequence, we decided not to use client-side encryption for now, and wait until a more wide-spread use of secure servers is in place.

The second monitor mode is the *maintenance* module, which implements a grade editor as well as basic addition and deletion of student accounts. Figure 13 shows a screen shot of the user account maintenance page, which allows the forced addition and deletion of students into the system. Since this requires special access privileges, the authentication database also is checked to determine if the user accessing the interface has the required rights to perform the editing operations.

The following is a list of actions (including queries) performed by the CGI programs after a user submits a grade edit form from the maintenance page.

1. Data is captured from the interface, and passed onto the CGI script. This includes login and password information.

2. The script checks against the database to see if the alleged user has the access permissions to change someone else's grade.

3. If this is the case, the records are presented for the student and a grade can be selected. If not, the system responds with an "access denied" message, and the process is over.

4. This data is once again passed back to the CGI program, which performs an "update" query in two different tables to update the score for that test and the overall cumulative score for the unit.

Figure 13: Add user page in the monitor module

5. After scores are updated, a message is sent back to the user indicating that the new information is now in place.

As we showed in Figure 13, users can be added and deleted using the maintenance interface. This also can be done interactively using a WWW-based interface. The next subsection provides an overview of the tools used to access the database, and a description of the activity and roster modules of QUIZIT.

4.3.2 mSQL and W3-mSQL

The database used by QUIZIT is mSQL, or mini SQL, a lightweight engine designed to provide fast access to stored data with low memory requirements [19]. As the name implies, mSQL uses a subset of the Structured Query Language (SQL) as its query interface. Although many SQL features are not supported, like "views" or "nested queries", mSQL provides the relational capability of multiple table joins, and is fully ANSI compliant. The complete mSQL package includes the database engine, a command-line interface, an administration program, backup support and a C language API. The engine is designed to work in a client/server environment using either TCP/IP or UNIX domain socket protocols. The distribution also includes source code, and is supported on many different UNIX platforms.

The mSQL daemon, msqld, is a standalone application that usually runs as *root* and that listens for connections on a well-known TCP socket. It is a single process, single threaded engine that will accept multiple connections and serialize queries. It uses

memory mapped I/O and cache techniques to offer rapid access to data in the database. In addition, it uses a stack-based mechanism that ensures similar performance speed on INSERT operations. Preliminary testing has shown that for simple queries, like the ones used by QUIZIT, mSQL performs better than other freely available database packages [19].

The API library included in the distribution, libmsql.a, enables any C program to communicate with the database daemon (in our case, CGI programs). Queries are submitted using a simple "msqlQuery()" call, and results are returned to a predefined C structure. mSQL also supports access control by using a ".acl" file in the installation directory. Permissions can be set for individual operations on a database, and restrictions can be placed on host names, for example. In addition, the ACL information can be reloaded at runtime using a *reload* command. This allows permissions to be changed automatically by another program, for example.

The three main reasons for choosing mSQL as QUIZIT's underlying database engine were the power of the C API, the wide variety of platforms it supports, and the ease of integration with a WWW server. In fact, several third-party contributions are available that provide query interface development for the WWW. The most notable of them probably is **W3-mSQL** a CGI script that extracts mSQL queries embedded from HTML code, and provides additional commands for presenting query results in HTML, without the need of using the C API. The use of W3-mSQL enabled us to quickly develop the monitor interface without a single line of C code. Figures 14a and 14b show the activity monitor in QUIZIT.

As we can see in Figure 14, the activity module in QUIZIT provides a report of all accesses to the system plus performance results on individual quizzes. In case the course is organized by units, it is possible to monitor only the activity for a given unit, if desired. As class size gets bigger, nevertheless, these activity logs may get quite large (possibly filling more than 20 pages long), and it makes sense to also have individual student activity logs instead of just entire class logs. Consequently, QUIZIT provides another interface for individual log activity. Figure 15 illustrates this interface, which contains an activity report for a given student in the class.

Figure 14: Activity interface in QUIZIT

Figure 15: Individual activity log

The final monitor module is the *class roster*, which enables instructors to check the progress status for each student in the class, including cumulative grades for each unit, the last quiz taken within that unit and the next quiz to be taken. This allows a more comprehensive view of each student's situation in the course, and a more direct way to compare their pace and performance. Figure 16 shows a screen shot of the roster module. Each student is represented by a *cookie*, a numeric code such as a social security number, for example. By using a code, instead of a name, in this page, results can be posted directly with a higher confidentiality, if needed. Assuming the course grader does not

have access to the authentication table where students' names are listed, this also hides this information during the log analysis process.

Figure 16: Class roster monitor interface

Section 4.4 describes the process that takes place when a student takes a quiz. We call the set of scripts that implement this functionality "the run-time environment".

4.4. The Run-Time Environment

The part of the system responsible for all login, grading and reviewing activity is called "the run-time environment" (RTE). The RTE is implemented as a set of CGI scripts written in ANSI C and compiled on the same machine running the WWW server (*httpd*, or http daemon) and the mSQL server (*msqld*, or mSQL daemon). The fact we have CGI scripts running on the same machine as the WWW server is a requirement, mainly because of performance issues: since CGI is a synchronous, transaction based protocol, having a CGI running on another machine would introduce a considerable bottleneck into the process, since yet another communication protocol would have to take place between the WWW server and processes running on different machines. For a similar reason, we also have the mSQL daemon executing in the same machine, as database transactions are tightly coupled with the scripts' functionality. In fact, we can think of QUIZIT's RTE as an extension to the WWW server that records specific state and context information by using an underlying relational data model. This type of architecture is very common on today's transaction based systems in the WWW [13, 19, 24], and has been proven to work well both in industry and academia [19].

Table 8 shows the different CGI, RTE processes running on QUIZIT. Each process takes care of a separate CGI transaction: the *login* process is responsible for first user authentication into the system, for fetching the first quiz page, and recording the begin time for that user into the activity's database. The *grader* process is responsible for grading the test submitted, updating the database accordingly, and providing the feedback information back to the student. The *mkcookie* process fetches the next quiz after the feedback page (e.g., a remedial quiz), reports whether a quiz is completed, and updates the database (begin time for the next quiz, current quiz being taken). The following paragraphs describe in more detail the implementation of each of these modules, with examples and illustration of a real setting. In addition, there are two supplementary modules that enable the student to review mistakes and old quizzes taken: *rpage* goes through students' mistakes and highlights questions that were answered incorrectly after the quiz was taken; *review* enables students to review old quizzes (i.e., quizzes that were already taken) and their progress in the course.

Table 8: QUIZIT's run-time processes

RTE Process	***Purpose***
login	Authenticates a user, and fetches the first quiz page. Also inserts the begin time and quiz information in the activity database.

grader	Checks each student's answers against the key generated by the QML. Grades the quiz accordingly and determines which quiz should be taken next. Updates scores, end time, and quiz information in the database and generates a feedback report.
mkcookie	Fetches another quiz after the feedback page, updates the database accordingly.
rpage	Highlights incorrect answers, one at a time, after the user follows a link from the feedback page.
review	Provides a comprehensive record of a user's activities and allows old quizzes to be reviewed (but not re-taken).

Figure 17 shows the login page. In addition to the unit's name, each user is asked to enter a password and a confirmation. This password/confirmation scheme was done because we are in the process of implementing client-side authentication using JavaScript. By performing an additional check we insure the login information passes a first authentication check before being sent to the server. For non-JavaScript compatible browsers, however, both the password and the confirmation are sent insecurely to the server, encrypted (using UNIX's crypt function, and a the first two characters of the encrypted version stored in the database) and checked against the encrypted password stored in the database. If the two are the same, and the user login is valid, the process queries the mSQL authentication database to obtain the user session code, or *cookie.*

After the cookie is retrieved, the unit name table is checked for the unit's code, and then finally the same code is used to access the unit log table containing the information about the quiz to be fetched for that user. After this information is read, the appropriate quiz is fetched back to the WWW server with the authentication cookie written in the HTML page as a hidden parameter. Then, the system performs another query to update the begin time, the current quiz, and also puts a "PROGRESS" flag in the end time to denote that the quiz is currently in progress (in case someone is monitoring the process using the monitor module).

The next process in the transaction sequence is the grader. After submitting a quiz, the grader obtains a list of question IDs and answers, and a cookie uniquely identifying the user. The grader then checks each answer against the key and generates a feedback page. The following list explains in more detail the actions performed by the grader:

1. The data submitted using the CGI POST method is parsed by the program and results are stored into an array.

2. According to the quiz information retrieved from the page, the grader reads the answer key generated during the QML compilation process, and stores each answer on a hash table.

3. The grader performs an update query in the unit's database, records the end time for the quiz.

Figure 17: QUIZIT's login page

4. Each answer in the form is checked against the answer database. Points are added or subtracted according to the original author's specification, which is also encoded during compilation inside the answer key file (extension ".ans").

5. The final score is checked against the cutoff mark. Then a second select query checks in the "quiz_log" table to see whether the quiz was previously taken. In case not, a third query updates the cumulative score for this unit by adding this quiz's score. In addition, the score for the quiz is stored separately and the NEXT field also is updated, reflecting which quiz should be taken next, based on the performance and the cutoff mark.

6. In case the quiz was previously taken, the score is not added to the cumulative score, and the next field remains unchanged, but the submission record is still logged into the quiz_log table.

7. A feedback page is generated (see Figure 18), containing a list of the questions incorrectly answered (if any), a score report, a time report, and a message specifying the next quiz to be taken, based on the performance of the quiz just submitted.

It is important to note that there are no time restrictions imposed by the system. It is left to the instructor to decide whether a student should be penalized for taking more time than necessary to finish a quiz. The reason for this design decision is the difficulty in determining whether a long submission delay is a consequence of the student's incapacity

to respond the quiz questions on time, or is due to network failures (e.g., failures in retrieving the exam or submitting it through the network). Further, as a consequence of this lack of time restriction, one can leave the system and come back at any point in time. The sequence of database queries assures the consistency of each student's records at any point.

Figure 18: QUIZIT's feedback page

The CGI script *rpage* allows students to review incorrect answers by following a link for each question. The target location is the quiz page, with the question highlighted. Unfortunately, the current version of the system only allows the highlight of one incorrect question at a time. In addition, we do not show the correct answers when reviewing quizzes using this mechanism. However, additional feedback always can be obtained by retaking quizzes as many times as necessary to figure out what the correct answers are. This can be accomplished by hitting the back key in the browser in the feedback page.

Mkcookie is the CGI script called when a student proceeds to take a next quiz from the feedback page (see Figure 18). This script is similar to the *login* script, recording the begin time for the next quiz, and updating the database records to indicate that a student is currently taking another quiz. Therefore, the process of taking quizzes in sequence can also be viewed as the execution cycle as illustrated in Figure 19:

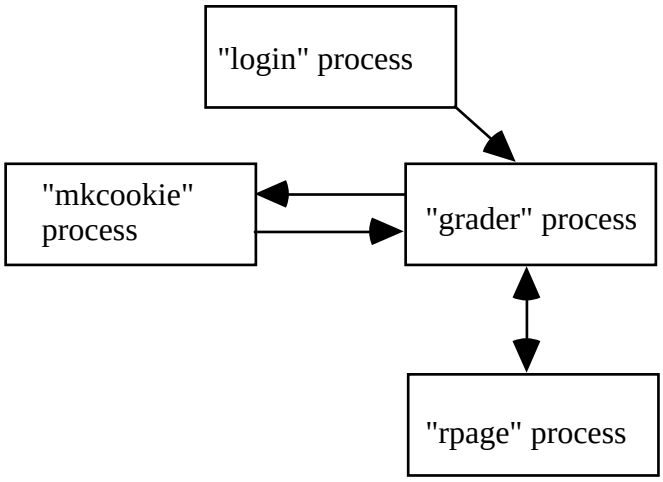

Figure 19: Sequence of CGI process calls during a test session

The final process in the RTE is the *review* process, which allows students to review their progress in the course and look at previously taken quizzes. The review page is illustrated in Figure 20. To access the review page, a student has to go through an authentication mechanism much like the one we described during the login phase.

In the review page, each quiz name also is a hyperlink to the HTML quiz page. Students reviewing old quizzes may not submit them for credit, but they may retake them for practice as many times as desired. Also note that the report format looks very similar to the individual activity page available in the monitor mode. Nevertheless, this time we did not use W3-mSQL to develop the script, because there are authentication features (e.g., encryption) that are not available in the W3-mSQL elements.

Figure 20: Review page

5. Testing and Case Studies

This chapter will describe QUIZIT's initial testing in three case studies during the Fall semester of 1996. We will further subdivide this description into two sections. The first section of each case study will analyze the instructor and teaching assistant's perspectives on the general use of the system, including usability issues, the time spent on authoring quizzes with QUIZIT, and the time savings in submission and grading of exams. The second subsection will explore the students' perspective. This will include critical incident reports via e-mail, student performance and progress in the course in accord with the Personalized System of Instruction (PSI) [10].

5.1 SIGIR '96: Preliminary Testing, Monitored Conditions

QUIZIT's early testing has related to the topic of "digital libraries", during a tutorial given at ACM SIGIR '96 in Zurich during August 1996. In this tutorial, a 104 page self-study was prepared to update materials used the previous year, along with a 6-part QUIZIT "quiz" (with 4-14 questions per part). This "quiz" follows the linear schema shown in Figure 20. Each module (e.g., a1, a2, etc.) shown in the figure below covered a distinct subtopic.

The test was administered in a lab equipped with one machine per attendee. The instructor's machine was a Sun Voyager running a local WWW server and the QUIZIT system. The set up was therefore of a typical local Intranet, i.e., a restricted domain sharing a WWW site.

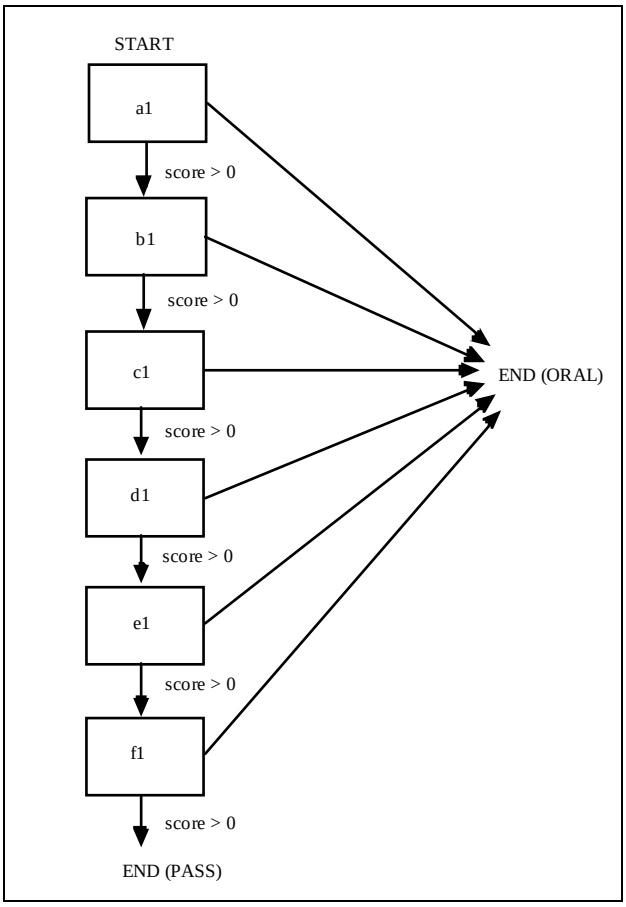

Figure 21: Structure of Digital Libraries Quiz given at SIGIR'96

5.1.1 Instructor's Experience

According to the instructor, preparing the quiz took approximately the same time as preparing a regular quiz, though about 4 more hours were required after that to make it operational. This included learning time, time to fix any problems related to compilation, hypertext presentation, and placement of generated files into appropriate directories, and additional time to register the attendees into the system (including individual passwords for the lab machines).

5.1.2 Attendees' Experience

Once the attendees started taking quizzes, however, very little time was spent answering questions about the system. They generally found QUIZIT easy to work with, and spent 15-30 seconds on average per question (i.e., 1 to 7 minutes per quiz on average). Since attendees were mostly novices, and the quiz questions were considered difficult, it was not surprising that scores were below 50%. However, this was done deliberately to motivate further study of the extensive tutorial notes [22].

After the conference, when activity logs were more carefully analyzed, we found that attendees who took the quizzes earlier and faster achieved on average better scores than students who procrastinated or took more time in each question. In addition, we found that some students actually accessed the system *after* the tutorial, from remote locations.

5.1.3 Problems Found

Most of the problems found during this first real test of QUIZIT related to installation and setup procedures. At that time, the password registration system had not been developed and the instructor had to assign individual IDs, cookies, and passwords for each of the 20 attendees. In addition, some usability problems with the QML were initially reported in the authoring process, during preliminary interview sessions with the instructor. Although problems with the QML still exist, as we will report in Chapter 7, these initial reports were very valuable to fix these first problems.

5.2 CS5604: Graduate-level course, PSI

The most comprehensive test of QUIZIT is being done in CS5604, an Information Storage and Retrieval course at Virginia Tech that has been offered in various forms since the early 1970's. In 1993 this masters level course was restructured according with the Personalized System of Instruction (PSI) [10], but with administration aided by gopher and e-mail. In 1994, course material was ported to the WWW. As reported by the instructor in [22], as record keeping and assessment in PSI often require a great deal of instructors' time, a special test administration system, using e-mail for test submission, was used in 1995. To further improve feedback, and reduce the increasing grading overload on the instructor, QUIZIT is being used starting in 1996 to automate the process.

In the next paragraph, we describe the test system and the evaluation schema introduced in 1995. In addition, we explain some of the problems with that system, and the motivation for the shift towards further automation in student evaluation. We then describe how QUIZIT is being used in conjunction with other methods of evaluation in the class.

The test administration system first introduced in 1995 allowed students to retrieve short essay answer, open book tests in the WWW by using a password registration system similar to the one now used in QUIZIT. However, test submission was done via e-mail and there was not any form of automated grading. There was one test for each of the 11 units in the course, and students could take these tests in any order. In addition, students were advised not to spend more than 90 minutes in each test, even though enforcing this

time limit was left to the instructor's discretion. The tests followed the schema shown in Figure 22.

Each test had three different versions for a same topic (e.g., a, b, c). Students who did not achieve a minimum of 90% on a test were required to retrieve and take a different version until they either passed or failed the third version, which would require them to report to the instructor for an individual oral examination. In this meantime, the instructor would receive activity reports in the form of e-mail messages sent by the system every time a student retrieved a test. Since test answers were submitted by e-mail, elapsed time could be calculated by comparing the retrieval and submission messages.

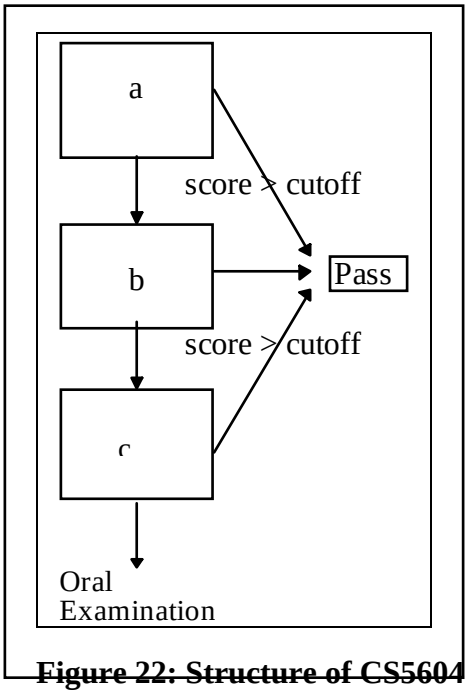

Figure 22: Structure of CS5604 tests in 1995

With approximately 20 students taking the course that year, one can imagine the overhead in grading and record keeping caused by the number of e-mail submissions that had to be graded and reviewed for each of the 11 units in the course. In addition, it was often the

case that, even with the help of a graduate teaching assistant, feedback would take one week or more, which in some cases disturbed students who were taking more than one quiz at a time and found out later they needed to take a second version of a quiz a week after their first, failed attempt. In fact, we can refer back to [18] to see the prejudicial effects of long feedback in self-paced instruction.

Figure 23 presents the new test schema in CS5604 using QUIZIT. Each unit is divided into two parts (a,b) along topical lines, with three versions per part. Students are encouraged to study more before attempting a different version, if they have trouble with one of the versions. Even though students will loose in terms of moving from instructor/teaching assistant evaluation of short essays in the old system, we expect they will gain from better organization of the questions into the adaptive framework of QUIZIT, and with instant feedback. In addition, since overhead in grading is greatly reduced, other types of evaluation can be introduced (e.g., class projects, more exercises) to further assess students.

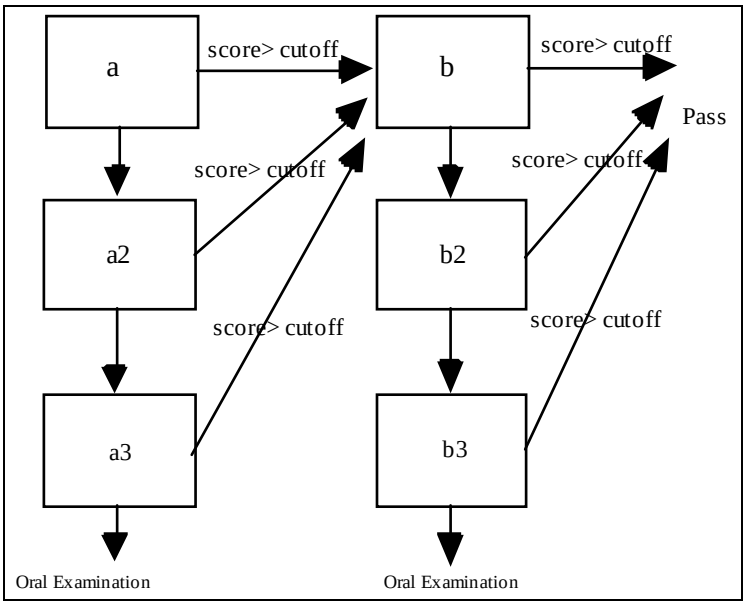

Figure 23: Structure of CS5604 tests with QUIZIT

5.2.1 Instructor's Experience

Compared to authoring short-essay quizzes, we found that the instructor took on average 3 extra hours for each of the 11 quizzes in the course, to get things converted to QML and then uploaded. There also was an additional time to have the two parts for each quiz and to convert to a multiple-choice format, instead of 5 short-essay questions (on average another few hours). The main advantage, however, is the reduction in grading time resulted from the use of QUIZIT, and the introduction of automated progress reports and record keeping. In addition to monitoring the rate of progress, the instructor can determine, for example, who seems to have taken longer or made more errors on a quiz, to facilitate tutoring and other assistance. In fact, it was often the case that the instructor in CS5604 intervened in the process to understand why some students were taking longer than necessary to complete a quiz. Further, as also seen in Figure 23, an oral examination may be required for students who fall through versions a3 or b3. This provides fairness to students who tend not to perform well on multiple-choice exams whereas it "filters out" students who do not have problems with this kind of examination. In CS5604, however, we found orals were necessary only in two or three cases throughout the semester.

A great benefit for the development of QUIZIT during the semester was to have a continuous feedback from the instructor concerning usability problems with the authoring or monitoring system interface. Many interview sessions were performed (most of them informally documented) and changes made accordingly. We found that, especially in the monitoring interface, ease of customization is crucial, as different classes and courses may have different evaluation structures and record keeping mechanisms. Nevertheless,

some of the monitoring procedures, however, like student registration and authentication, activity report, can be generalized.

5.2.2 Students' Experience

The most interesting analysis, however, comes from observing the QUIZIT log files to determine how students use the system. For CS5604, two types of activity analysis were performed: *a progress analysis*, which answers questions like "do students procrastinate when taking quizzes in PSI?" or "how does the fastest student progress during the course?"; and a *performance analysis*, which explains how well students did on different quizzes in terms of the number of versions (levels) required to pass a certain unit.

Table 9 shows the percentage of students (relative to a universe of 11 students finishing the class) who completed a certain unit within a certain number of weeks. The numbers 1,2,3,4, and 4+ refer to the number of weeks after the lecture on that unit's topic. 4+ should read "more than 4 weeks", and "B" means "Before the lecture". For example, assuming the lecture for the topic "DL -- Digital Libraries" was given on August 27, 1996, the percentage under column "2" means that 27% of the students completed the test before September 10, 1996 (i.e., within 2 weeks after the lecture). Note that the percentages are cumulative, so the entries for column "4+" should always read 100% unless one or more students failed to successfully complete a test.

Table 9: Progress report in CS5604 during the Fall of 96

Unit\W	**B**	**1**	**2**	**3**	**4**	**4+**
DL	0%	0%	27%	72%	72%	100%
IR	0%	0%	18%	37%	65%	100%
IF	0%	0%	0%	18%	27%	100%
SS	0%	0%	0%	9%	18%	91%
RR	0%	0%	0%	9%	18%	100%
CL	0%	0%	0%	9%	9%	91%
IN	0%	0%	0%	0%	18%	91%
SD	0%	0%	0%	0%	0%	91%
HT	0%	0%	0%	0%	0%	91%
MM	0%	0%	0%	18%	91% (**)	91%
KB	0%	0%	9%	91% (**)	91%	91%

The data in Table 9 shows that students tend to procrastinate more during the middle units: the “average” student takes approximately 3 to 4 weeks to complete the first two units, then more than 4 weeks in the 7 middle units, until the end of the course deadline forces him or her to submit the last two quizzes. Looking into the actual activity data directly from QUIZIT, we also could see that 63% of the students (7 out of 11) had taken more than 3 quizzes within 48 hours of separation during the last two weeks of the course. This supports our assumption that this clustering of activity is caused by the deadline to submit quizzes. Further, we noted from the logs that the fastest student took

on average 3 weeks to complete each quiz, and that no one took quizzes within 1 week after the main lecture for a unit.

Table 10 shows the performance results for each unit. The column “level” indicates the number of versions of a quiz. For example, if a student takes version “a1”, then “b1”, then passes, we count 2 levels. On the other hand, if someone takes version “a1”, then “a2”, “a3”, “b1”, and then passes, we count 4 levels. We can therefore correlate the number of levels to a performance indicator, since the two threads “a” and “b” represent different topics within each unit. The best performance is represented by 2 levels (“a1”, then “b1”, then pass), whereas the worst performance is indicated by the column FAIL (more than 6 levels). Percentages are relative to the total number of students taking a particular quiz. So, if 1 out of 10 students took quiz RR, that shows as 10% in the Table; if 1 out of 11 took quiz SS, that in turn will show as 9% in the Table.

Data from Table 10 suggests that most students go through the quizzes in just two levels. Quiz IF is an exception: most students required at least 3 levels to pass it. Curiously, the logs do not suggest that procrastination leads to worst performance. In fact, we can almost say that there is an inverse relation between the two: students who take quizzes in clusters tend to perform better than students who do not. A possible explanation is that students do not proceed to a next quiz until they are sure they mastered the contents of a previous quiz, or the fact they have taken a hard or long quiz (i.e., with more questions) somehow scares them to take the next one. In any case, with just 11 students finishing this class, we lack enough data to conclusively support any of these theories with enough confidence.

Table 10: Performance data for CS5604

Unit\Lev	**2**	**3**	**4**	**5**	**6**	**Fail**
DL	91%	9%	0%	0%	0%	0%
IR	82%	9%	9%	0%	0%	0%
IF	36%	55%	0%	9%	0%	0%
SS	63%	0%	28%	0%	0%	9%
RR	60%	20%	10%	0%	0%	10%
CL	91%	0%	0%	0%	0%	9%
IN	91%	0%	0%	0%	0%	9%
SD	66%	25%	0%	0%	0%	9%
HT	50%	33%	0%	0%	0%	17%
MM	31%	30%	30%	0%	0%	9%
KB	0%	91%	0%	0%	0%	9%

We also note the steady percentage of "fail" marks. This can be explained by the fact that one student took an incomplete grade and just did not take all the tests. In only two cases, there was a requirement for an oral exam.

5.2.3 Problems found

Throughout the semester, we set up an online problem report mechanism through e-mail (quizit@ei.cs.vt.edu account). After cataloging the messages, we divided them into 3

different categories: problems with the course material, system bugs, and usability problems. Table 11 shows the percentages of problems found in terms of the number of e-mail messages sent to the QUIZIT account.

Table 11: Problems reported to "quizit@ei.cs.vt.edu" in CS5604

Problem category	***Percentage of e-mail messages***
Course Material	10%
System Bugs	35%
Usability Problems	55%

Most usability problems were suggestions for interface improvement in the system. The most common request was for improvement in the feedback page. Students often felt frustrated when the system failed to provide a clear explanation on why they missed a question. Unfortunately, that was often the case in the first version of QUIZIT, in particular with reporting correct answers for fill-in questions. This problem was fixed in the next version of the system. We will describe some of these problems in more detail in Chapter 6, where we present an improvement plan for future versions of the system.

Systems bugs often were reported from the review page or feedback page, and were fixed as they were reported. In version 1.0, we did not have a review page at all, so students often complained about not being able to review their own progress in the course nor the quizzes they had taken previously. When we finally added the review interface shortly after in version 1.1, alpha testing was still being done, which caused a lot of useful bug reports!

5.3 CS1604: Large enrollment, undergraduate level course

Initial testing of QUIZIT also has been performed in a large enrollment (approximately 160 students), one credit service course for incoming freshmen: CS1604, “Introduction to the Internet”. This course is further characterized by having diverse student population (i.e., majors and non-majors in Computer Science). All course material is available over the WWW at “http://ei.cs.vt.edu/~netinfo/”. Prior offerings of the course required students to attend lectures (presented with a WWW browser accessing online course material), submit assignments via e-mail (also graded through e-mail) and take in-class tests. There are two sections of the course: a traditional lecture version (with 120 students) and an independent study version (with 37 students). The instructor plans to further evaluate the course, quizzes, and test, using these experimental groups [22].

Due to the larger enrollment requests for this course and limited resources, there is a shift from manually-graded evaluation to automated evaluation and PSI for independent study. QUIZIT has been adopted during the Fall semester of 1996 to help attain these goals. Traditional assignments are gradually being replaced by online quizzes. Course assignments are no longer mandatory or graded. Instead, results from assignments will be incorporated into online QUIZIT quizzes to evaluate assignment knowledge gains. Immediate feedback is expected to compensate for the increasing delays resulting from the old grading system. Quizzes are published bi-weekly, and students are encouraged to proceed at their own pace, in accord with PSI.

5.3.1 Instructor's Experience

As in CS5604, an online training program was developed to introduce the process of authoring quizzes using QML. Again, formal interviews were conducted and preliminary critical incidents tabulated. We found that, once trained, the instructor took about the same time to author quizzes using QML (approximately 4 hours). The incidents found during the training phase served as preliminary bug reports for the developer. The bugs found were corrected before the first quiz for CS1604 was published.

Tag recall (i.e., remembering the correct tag name once the desired functionality is determined), question naming (i.e., incorrectly assigning duplicate IDs to question items) , and point assignments (assigning scores to different answers for a question) were the main problems. To solve the tag recall problem, a QML template was made according to the quiz structure used in the class (we will describe this structure in further detail later). The question naming problem was solved by further improving the compiler's error messages, and with more training. The biggest problem, however, was with point assignments. Currently, QUIZIT does not provide multiple-choice questions with radio buttons. The scheme used requires instructors to assign negative points to incorrect answer options, as to prevent users from selecting all the answers and still get points. For example, assume you have the following question:

```
<MCHOICE ID="fingersinhand">
<DESCRIPTION><P>How many fingers a normal human person has in a hand?</P>
</DESCRIPTION>
<ANSWER ID="q0001" VALUE=WRONG POINTS=0 ><P>1 finger</P></ANSWER>
<ANSWER ID="q0002" VALUE=RIGHT POINTS=5><P>5 fingers</P></ANSWER>
```

```
<ANSWER ID=“q0003” VALUE=WRONG POINTS=0><P>4 fingers</P></ANSWER>
</MCHOICE>
```

Since radio buttons are not available in multiple-choice question (which would allow the selection of only one item), a student who selects the three boxes will eventually get the five points assigned for correct answer “5 fingers”. A solution for this problem is to assign negative points for incorrect answer items, so a student can be penalized for selecting incorrect answer(s). Nevertheless, we found that this solution confuses both instructors and students, which are not used to such a negative scoring scheme. A next version of the system will have a type attribute inside the `<MCHOICE>` GI called “`SELECTONE`”, to cover the frequent case of having just one correct answer among many possible choices, in multiple choice questions.

The quizzes in CS1604 have the linear structure shown in Figure 24. Students are advised to take quizzes in order, although this is not required. There are only 6 quizzes for the entire course, although each tends to have many more questions than the quizzes in CS5604. The structure is flat, which means that quizzes do not explore the adaptable capabilities of the system. Each quiz is worth 100 points, and scores are used directly in the computation of grades.

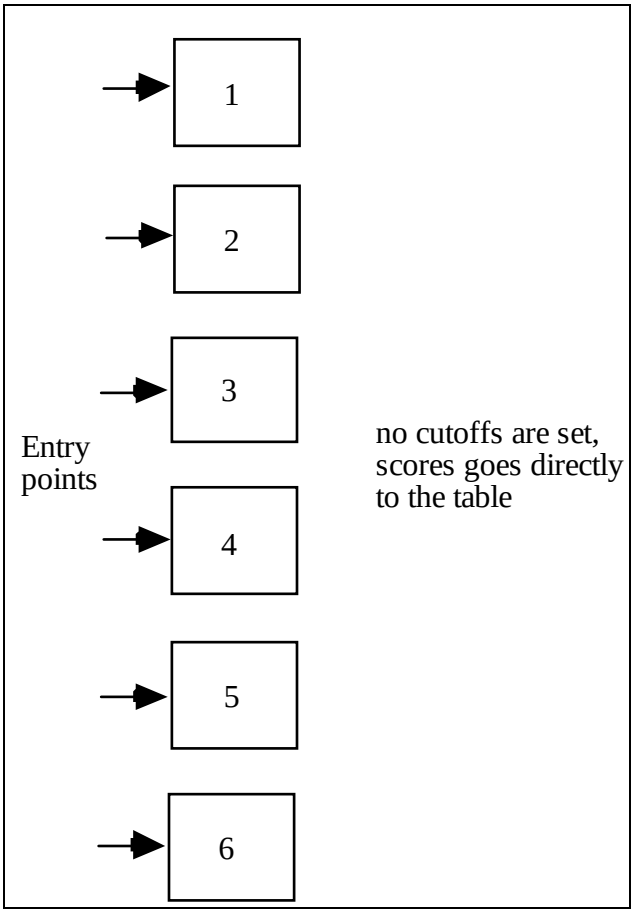

Figure 24: Structure of the quizzes in CS1604

5.3.2 The students' experience

An e-mail "hotline" was available so students could report their problems with the system to the GTA, the QUIZIT developer, and the instructor. Their main problem appeared to be with registering their passwords and first logging into the system. As some students were not even familiar with a computer, there were a few cases where it was hard to distinguish between usability problems with the WWW browser and usability problems with QUIZIT. Most of these problems were solved once we improved the error messages in the system, especially the ones related to the login/ registration procedure (e.g., "Please check to see if you really typed in your social security number without dashes", etc.).

Other problems involved reviewing their scores: once again we attested the importance of providing a full feedback report including their answers, the correct answers, and the number of points deducted when the two did not match. Once we improved these reports, questions were limited to the course material.

As in CS5604, we also used the monitor interface to keep track of student progress during the semester. We found that students also procrastinated in CS1604, but to a greater degree. Table 12 shows a progress report for the class. The cumulative percentages refer to the number of students taking a test within the number of weeks shown. For example, the percentage 10% on row 1 and column 2 mean that only 10% of the students registered for the class took quiz "2" within 2 weeks after the subject for that quiz was taught. The column "B" refers to the quizzes taken *before* the class for that subject.

Table 9: Progress report in CS1604 during the Fall of 96

Quiz\W	**B**	**1**	**2**	**3**	**4**	**4+**
1	0%	5%	10%	15%	50%	100%
2	0%	0%	0%	15%	35%	100%
3	0%	0%	0%	18%	27%	100%
4	0%	0%	0%	0%	18%	100%
5	0%	0%	0%	0%	18%	100%
6	0%	0%	0%	0%	5%	97%

As in CS5604, we see that students tend to procrastinate less in the beginning of the semester. From the activity logs in the database, we also see that they take quizzes in clusters close to the final submission deadline.

5.3.3 The developer's experience

It is worth mentioning that the use of automated grading shifted the role of the graduate teaching assistant (GTA) from merely a grader to an active developer of the system. It is clear that it is impossible to generalize QUIZIT's monitor interface to all possible uses in different classes. It is still a major task to initially tailor the system to specific record keeping needs of a particular instructor, and to provide the technical support for all the students using it in the class. We were fortunate to find out that these tasks were much more rewarding for the GTA than the traditional tasks of manually reviewing hundreds of assignments using e-mail.

6. Future Work

This work is certainly not finished: there are still major improvements to make QUIZIT a more stable and usable product in the future. In this chapter we list several of these suggested improvements in a prioritized order, so future developers and project sponsors can have an initial plan to follow:

1. Training Program (1 -- 2 weeks):

Since the developer's documentation online is still very poor (limited only to inline comments in source code), there needs to be an initial training program for the developer(s) undertaking the development program of QUIZIT. Initial training can be provided by Mr. Tommy Johnson for this purpose.

2. Tasks (first item has highest priority)

2.1 Make QUIZIT portable and distributable (2 weeks)

A version control program needs to be put in place. In addition, a more careful analysis of portability issues needs to be done (e.g., how to configure the system for different WWW servers on different machines) so system installation can be fully automated.

2.2 Improve the documentation (3 weeks)

At this point, online documentation is available only for the authoring interface (http://pixel.cs.vt.edu/~ltinoco/quizitdocs/quizit.html). Other tutorials, especially for a developer's manual need to be done. Further, it is important that further changes in the system are published to the outside world as well (see references for papers online).

2.3 Code and Database redesign to C++ or JAVA (4 -- 8 weeks)

Recoding in C++ or Java would provide a higher degree of code reusability and abstraction, especially of the mSQL dependencies, thus creating a more general database model (in case Java is used, one can choose use the newly developed ODBC-Java class library). Redesigning the database would eliminate some of the redundancy that still exists in the tables. In addition, the use of remote method invocation through distributed objects in Java may be an alternative to CGI, although the issue of reduction in portability has to be carefully considered.

2.4 Solve small bugs and usability problems (8 -- 16 weeks)

In the compiler (4 -- 8 weeks):

• Improve the grammar (QML) as to reduce the need of using paragraph tags where they can be omitted without penalty. To accomplish this, changes need to be done in the YACC code, the code generator, and the documentation.

• Provide a better naming schema for questions IDs, especially in matching questions (to do this, changes are also necessary in the code generator).

• Create an attribute for multiple choice questions to allow a “select one only” option with radio buttons (this was mentioned in section 5.3.1)

• Allow fractional points to be assigned to question items, so a more general scoring technique can be applied. This should be easily accomplished by changing the grader engine.

• Create additional attributes for fill-in questions to allow more powerful pattern-matching options. One can use the built-in C library routines for pattern-matching to accomplish this.

• Create another question type, possibly called “SHORTESSAY”, to enable students to enter short-essay answers that can be mailed to the instructor for future e-mail feedback. This requires another CGI script to handle these answers in the server-side and

additional changes in the compiler to generate another HTML form, and translate the additional tags.

• Create a QML visual constructor (or “wizard”, in Microsoft terms), possibly using JAVA and/or HTML templates and forms.

In the run-time and review engines (2 -- 4 weeks):

• Provide a better feedback report, possibly including correct answers in the quiz being reviewed. Right now, the error report module only shows the errors, and not the solutions. Additional database queries may have to be done in the grader engine.

• In the review mode, create the option of reviewing all the quizzes in a given unit, once it is already cleared by a student. This should be done by an additional query in the review engine, searching for all the quizzes for a given unit.

• Provide a pattern-matching module in the grader for more complex fill-in questions. This task can be greatly simplified if we use the pattern-matching C++ libraries available today in most UNIX platforms.

In the monitor engine in w3-mSQL (2 -- 4 weeks)

• Generalize the model for any class (e.g., what if the instructor wants to have a different unit structure in the course?) Provide tools for the construction of this model, such as a GUI for the automatic construction of the class database and a more general database editor (i.e., and not just for grades).

• Enhance the security mechanisms, possibly using client-side encryption in Java.

7. Conclusion

As stated in Chapters 1 and 2, automatically graded multiple choice exams are certainly not the solution to the bigger problem of evaluation in underbudgeted, large-enrollment classes. Nevertheless, several works in the literature [2, 8, 15] describe that multiple-choice exams, especially if automated, can be quite effective in **supplementing** other forms of evaluation. In particular, these papers suggest that multiple-choice tests can be a quite successful mechanism in "filtering out" the "good" students (who are also good test-takers) in a class. Therefore, if our goal is to reduce grading overload, this method of evaluation can be quite effective when combined with other types of evaluation like oral exams, even in large classes. If we assume that on average 70% of the students will go through these automated tests with no problem, we are left with only 30% of students who need to be more carefully evaluated (either by taking a conventional paper exam, or an oral, for example).

Our recommendation is therefore to use QUIZIT as in CS5604, whenever possible. Our experiments found that test automation combined with oral examination and other projects worked quite nicely on that case. In addition, it is highly advised that instructors experiment with adaptable tests as much as possible. This capability provides the highest degree of interactivity, and performance adaptability (when well planned and designed) and students tend to like it better (we actually interviewed students in CS5604 to support this). Besides, it is much more effective to use opscan sheets to grade exams with a flat file structure like the ones used in CS1604 (except when multimedia content is included on the quiz). To address the problem of procrastination, we suggest that bonus points be given to students who finish their quizzes within 2 weeks of the target lecture. This way

we can keep the self-paced characteristics of the course (since we do not penalize them), but also motivate students to create the discipline for a more steady pace in the course.

The experience in CS1604, however, has given us the opportunity to study a different methodology for class administration. We saw the shift in the role of the GTA from just a grader, to a *monitor* , and a developer. Our view is that in classes using QUIZIT, we can concentrate our grading efforts not only on more personal types of evaluation, but also on improving QUIZIT itself!

References

1. Asymmetrix. *Toolbook CBT Edition*.. [http://www.asymmetrix.com/], 1996.

2. Belnap, N.D., *The Logic of Questions and Answers*. New Haven. Yale University Press, 1976.

3. Bryan, M., *SGML: An Author's Guide to the Standard Generalized Markup Language*. New York. Addison-Wesley, 1988.

4. Coombs, J.H., A.H. Renear, and S.J. DeRose, Markup Systems and The Future of Scholarly Text Processing. *Communications of the ACM*, November (1987), 933-947.

5. ETS, *Registration Bulletin: GRE General Computer Based Test Descriptive Booklet*. Princeton, NY. Educational Testing Services Inc., 1995.

6. Fox, E., *Interactive Learning with a Digital Library in Computer Science*. Virginia Tech, 1996.

7. Goldfarb, C.F. *A generalized approach to document markup.* In *ACM SIGPLAN-SIGOA Symposium on Text Manipulation*. Portland, Oregon 1981. ACM. New York, 68-73.

8. Hansen, D.N., ed. *An Investigation of Computer-Based Science Testing.* Computer-Assisted Instruction: A Book of Readings, Ed. R.C. Atkinson and H.A. Wilson. Academic Press Inc. New York, NY, 1967.

9. ID2 Research Group, U.S.U. *The Electronic Trainer*. .

10. Keller, F.S., Good-bye, teacher... *J. of Applied Behavioral Analysis*, 1, 1 (Spring 1968), 79-89.

11. Levine, J.R., T. Mason, and D. Brown, *Lex & Yacc*. 2nd ed. Sebastopol, CA. O' Reilly & Associates, Inc., 1992.

12. Lindstrom, S. and J. Violette. Gearing Up for Training: The Nuts and Bolts of CBT Authoring. *New Media* 6, 13 (Oct. 7, 1996), 37-47.

13. LRCorp. LXRTest. [http://www.lxrtest.com/], 1996.

14. Macromedia. *Authorware CBT*. [http://www.macromedia.com/authorware].

15. Mason, R., *Using Electronic Networking for Assessment*, in *Open and Distance Learning Today*, F. Lockwood, Editor. Routledge. London, 1995. 208-217.

16. McManus, T.F. Delivering Instruction on the WWW. [http://www.edb.utexas.com/coe/depts/ci/it/projects/wbi/wbi.html], 1995.

17. Nichols, G. Greg's Web Based Training Place. [http://www.ucalgary.ca/~gwnichol/index.html], 1996.

18. Rekkedal, T., The written assignments in correspondence education: effects of reducing turnaround time. *Distance Education*, 4, 2 (1983), 231-252.

19. Rowe, J., *Building Internet Database Servers with CGI*. 1st ed. Indianapolis. New Riders, 1996. 395.

20. Tinoco, L., *et al.* QUIZIT: An Interactive Quiz System for WWW-based Instruction. [http://ei.cs.vt.edu/QUIZIT.ps] (Postscript format), 1996.

21. Tinoco, L.C. QUIZIT Authoring Tutorial. [http://pixel.cs.vt.edu/~ltinoco/quizitdocs/], 1996.

22. Tinoco, L.C., E.A. Fox, and N.D. Barnette. *Online Evaluation in WWW-based Courseware*. In *SIGCSE '97 Technical Symposium on Computer Science Education*. San Jose, CA, Feb. 1997. ACM.

23. Turner, R.C., T.A. Douglass, and A.J. Turner, *README.1st -- SGML for Writers and Editors*. Goldfarb, C. F. In Open Information Management, ed. C.F. Goldfarb. New Jersey. Prentice Hall, 1996. p.241.

24. Wheeler, D.A. MkLesson User Guide. [http://lglwww.epfl.ch/Ada/Tutorials/Lovelace/userg.html], 1996**.**

VITA

Lúcio Cunha Tinoco was born on a warm day in the beautiful city of Rio de Janeiro, Brazil on September 18, 1971. After studying music, jazz composition, and computer engineering in Brazil, he transferred to University of North Carolina at Chapel Hill, where he received his Bachelor of Science in Mathematical Sciences. After graduation, he joined the masters program at Virginia Tech. Besides music and soccer (as any other "normal" Brazilian), his main interests are Information Storage and Retrieval and Internet Software Development.