ITEC810 Final Report

Randomised Quiz System for Assisted Learning

By Benjamin Evans - 40729052

Project Supervisor: Christophe Doche

Prepared October 2010

Table of Contents

Abstract

A common objective in Education institutions is to facilitate the learning of subject material with the aid of computers. Creating on-line quizzes to test student understanding and to encourage self-learning of important subject concepts has been well proven to boost student results in tertiary education. Most quiz systems rely on randomly testing questions from a “pool”; each question and answer in this pool having previously been specified by an educator.

In the late 1990‘s, prototypes of randomised quiz systems in both Mathematics and Physics began to emerge. Each quiz generated from these systems were unique, as both questions and answers were randomly generated from pre-defined formulas. They were also self-marking. This allowed educators to help assess the level of understanding of various concepts with ease.

In this report, I provide a detailed presentation of the implementation of a similar randomised quiz prototype for university students studying programming - specifically C and C++. Previous and similar attempts at such systems are outlined and contrasted. This prototype is called The *Randomised Quiz System for Assisted Learning.*

The prototype is an extensible system in which students are presented with a set of randomly generated programming questions, based upon concepts that educators have previously defined and wish to assess. The system will grade the students based on their responses and give appropriate feedback. This helps students understand and practice concepts, and gives the educator an idea of what concepts are actually being understood. Ultimately, it aims to help facilitate self-learning in the computing field.

Keywords: Computing, Assisted Learning, Web-based learning, Online Quiz

1. Randomised Quiz Introduction

Quizzes and examinations have been a part of learning for many years. With the advent of computers, many quizzes have been moved “online”. Online quizzes are now traditionally used for both assessment and self-learning [3][5][7]. They are also beneficial to students, as they allow them to self-learn with ease. Instant feedback with online quizzes also has the potential to help students learn faster than with traditional methods [8].

The concept behind a *randomised* quiz is that questions are generated by computer. The questions are based upon templates that are specified, and the random aspect ensures that students aren’t given the same question. This allows students to practice the same quiz repeatedly and get different answers. It also limits plagiarism.

Several randomised quizzes exist in fields such as Mathematics and Physics. The *Randomised Quiz System for Assisted Learning* aims to bring randomised quizzes to computing - specifically to programming courses.

1.1 Online Quiz History

When average households were gaining access to the internet, various schools and universities began developing web-based quiz systems. These were essentially an evolution of paper quizzes and consisted mainly of simple multiple choice questions. As time progressed, educators added more questions to these quiz systems and eventually had a set of questions for each subject. When the pool of questions became large, quiz systems could choose a random question related to a particular concept or subject from the pool, thus giving the effect that the quiz was randomised. This is still the method used by many online quizzes and learning management systems today [13]. Having these quizzes as part of a course generally boosts student results [11].

Later on, the concept of having the computer actually *generate* the questions came about. This was done primarily in Mathematics (as computers could use a formula to generate an appropriate question) [2][7][8]. This allowed educators to specify a single template and generate very large amount of questions from it. This is what is referred to throughout this report as a “randomised quiz”; It has since been adopted to subjects like Physics, and to a lesser extent Computing [3][5].

1.2 Problems with existing solutions

A problem with online quizzes that have static questions is the fact that students can often get the same question presented to them in one or more quiz attempts. If the same question is repeated when the student is re-taking a quiz, they will already know the answer. This defeats the purpose of self learning as they are not memorising *how* the answer was obtained, but rather *what* the answer was. Since questions are the same for all students sitting the quiz, there is the potential for plagiarism.

The only way to combat these problems in non-randomised (static) quizzes is to increase the amount of questions in the ‘pool’. Unfortunately, this is very time consuming as each question and answer must be manually written by educators.

A more effective method is randomising the questions themselves (rather than the order in which they appear) so that the chance of a two students getting the same question is very low. This is the motivation behind randomised quizzes.

1.3 Outcomes and Aims of prototype

The initial intended outcomes for this project involved writing a report on what questions are suitable for a randomised quiz system and delivering software that is able to *generate* randomised quizzes.

After commencing work on generating randomised quizzes, it became clear that extending this software so that it could be used in a University Environment would not be difficult.To do this, a more complete system would have to be developed that could not only generate questions, but assess students, give them feedback and allow lecturers to monitor student progress.

Consequently, the *Randomised Quiz System for Assisted Learning* aims to be a prototype that implements a randomised quiz system suitable for undergraduate university students studying both the C and C++ programming languages. The prototype will automatically generate questions from 'templates' that are specified by lecturers and will give students the ability to answer these generated questions online and receive immediate feedback in regard to whether the answer was correct.

The prototype is to be trialled at Macquarie University in 2011. Implementing this prototype in this environment aims to promote self-learning, help lecturers understand which concepts students are finding difficult and boosting students' overall understanding of concepts being tested.

1.4 Report Outline

The remainder of this report outlines previous works that are relevant to this project, the steps taken and decisions made during the design of the prototype, the back-end and front-end implementation of the prototype (including a critical analysis of the work done) and an evaluation of how successful the project was and any future work that could take place.

2. Related Works

The Randomised Quiz System for assisted learning is not the first of its kind. There have been many studies that outline the motivation for self-learning systems such as this. There have also been various attempts at creating randomised quizzes in computing, mathematics and physics; all have had fairly promising results. These works and studies are outlined and contrasted to *The Randomised Quiz System for Assisted Learning* in this section.

2.1 Motivation behind Online Learning and Online Quizzes

Online learning is a relatively new concept that came about with the widespread accessibility to the internet. It differs significantly from traditional lectures where an educator serves to deliver information to many students. Studies outlined in [9] argue that current-generation students are finding traditional lectures less effective than the generation before them and that students achieve more when learning 'actively'. The study argues that computer assisted learning environments, which are focused around interactivity appear to be more effective in allowing students to enter a psychological state in which they are efficient, intrinsically motivated and more likely to complete tasks assigned to them.

The idea of having 'interaction' in university courses is not new. Many courses offer practical sessions where there are instructors [6]. In studies outlined in [4], it was discovered that many computing students were self-learning and relying on practical instructors to serve as "*facilitators of personalised learning rather than broadcasters of knowledge*" [4]. Although instructors are proven to be quite effective in relatively small classes, in larger classes, the ratio of instructors to students can be too low and instructors aren't able to deliver personalised feedback to all the students that require it [4]. Instructors and lecturers should ideally give individual feedback on weekly exercises and all practical tasks, but in large groups this can sometimes be too time consuming [10]. This is an area in which online learning can help.

In subjects like computing, online learning could be seen as an evolution of traditional practical sessions. Although they obviously do not offer the same type of interaction, many online learning facilities do provide instant feedback, and some studies have shown that generally students are cautious but approving of online learning. In a study outlined in [1], it was seen that teachers rated textbooks to be one of the least beneficial learning resources. A paper analysing traditional learning methods and comparing them to online learning [12] argued that students prefer to learn by concrete example and experiments. [12] also argues that students learn effectively by observing multiple examples and making their own relevant conclusions. Labs and practicals were seen to be effective, but students particularly enjoyed the instant feedback provided by online learning (specifically quizzes). It was then concluded that multiple-attempt quizzes drive "*reading, experimenting and abductive reasoning*" [12]. Online quizzes were also seen as an effective way to bring all students up to a certain level of learning [8].

Unfortunately plagiarism is also a problem in some subject areas. In one study [11], plagiarism was common in students' weekly practical work and assignments. This plagiarism was thought to

contribute to low grades. In this case, implementation of online learning and randomised quizzes appeared to combat the plagiarism problems and subsequently improve marks.

It can therefore be concluded that the main motivations behind online learning are to help prevent plagiarism, provide individual and personalised feedback to students that otherwise wouldn't receive it and to help encourage the self-learning process that is proven to stimulate motivation, reading and understanding of delivered content.

2.2 Existing Randomised Quizzes

Several online quizzes have already been trailed in educational environments with mixed results. This section outlines some of the more interesting attempts at online quizzes and their perceived advantages.

An early attempt at online quizzes is detailed in [11]. In this implementation, computer science students were assigned pre-written questions to complete in addition to weekly practical tasks. The initial student response was quite negative, but the eventual results yielded promising exam results. This implementation employed UNIX as the 'quiz system' (only accessible on campus) and had automarker programs to automatically grade student responses. Analysing this, an obvious disadvantage is that the students had to complete the quizzes on campus. There were also numerous technical faults with the implementation which led to some students being disadvantaged. Regardless of these disadvantages, the results obtained from implementing the quiz were promising.

Another interesting approach towards online quizzes is outlined in [5]. This quiz system was to help evaluate students studying SQL (a database querying language). It employed a more modern approach and was web-based. The interesting part about this implementation is that marking was done using a combination of automarking, peer review and lecturer input. Students would initially use another system that would show them how their answer looked instantly. This allowed them to tweak their answer until satisfied with the result. Peers in the class would then review other students' work online and give a recommended mark. This mark would be finalised by the lecturer. Unlike other systems this system rewarded a mark based on 'correctness', rather than the right/wrong marks given by other systems. The advantages of this implementation are that the marks given are not allocated by a computer, but rather by people. Another advantage is the feedback that is given to students is written by people. However. this in itself is a disadvantage as students don't get instant feedback and the marking process involves the time an effort of both students and educators.

A different approach that was taken in a Randomised Quiz system developed for Mathematics is outlined in [2]. This system used a backend "engine" to generate questions. This engine was *Matlab*, a program particularly suited to generating and solving mathematical equations. The engine is very mature, and this implementation involved templates and randomising certain variables to create randomised questions. The system would output an XML file that contained randomised questions, which could then be imported into Learning Management Systems like *Moodle* [13]. A similar approach was taken in [7][8] where instead of *Matlab*, Perl was used to not only generate questions and answers, but the intermediate steps taken that show how that answer was obtained. This implementation went one step further and used CGI, LaTeX, Javasciript and PDF technologies in

order to create an electronic quiz environment in which students could attempt quizzes (both multiple choice and 'fill-in the answer' questions), check their answers and obtain feedback on how to arrive at the correct answer if they got it wrong. The only perceived disadvantage of this method is using PDF modules (such as Javascript) that are not supported uniformly across different platforms.

Lastly, a reasonable attempt at a randomised quiz system for programming has already been developed - its implementation is outlined in [3]. It uses a compiler as the quiz engine and uses a system of parameterised substitutions to make questions random. The quiz interface is in HTML, and is limited to comprehension questions where a student is given a section of code and asked a question about it. The main advantage of this system is that it is completely automatic and randomised. Disadvantages of this implementation include a lack of feedback and a lack of the types of questions that can be asked. Anti-plagiarism mechanisms also appear to be lacking.

2.3 Impact on Student Results

As mentioned in Section 2.1, a motivational factor behind Online Learning is to help increase grades. This section briefly outlines the impact on student results that some of the quizzes outlined in *Section 2.2* had.

In [11], it was found that with the introduction of online quizzes, drastically decreased failure rates ensued. It was also found that exam marks correlated very strongly with quiz marks. A comprehensive survey was undertaken after students had completed a final semester which used a newly implemented quiz system.

The surveys in [11] found that students generally wanted online quizzes (54%), but around 41% of them wanted the marking to be done solely by humans; another 34% said that they wouldn't mind a combination of human and machine marking. This particular Randomised Quiz implementation offered no second attempts at questions (or at quizzes) and no feedback. One could deduct from this that students like the idea of Online Quizzes, but want feedback that is personalised in some way. Regardless of the marking style, 58% of students surveyed believed that online quizzes were a good idea and should be performed in the future.

In [3], a randomised quiz system was trialled over several years. The system (QuizPACK) was not used for mandatory assessment. However, very detailed results analysis was performed on the effectiveness of the system, and it was found to greatly benefit students who used it. The students who actively engaged in using QuizPACK performed better in in-class quizzes. It was also found that those who attempted the quizzes more often (and eventually achieved better quiz marks) generally performed better in the final exam.

In [7][8], a mathematics quiz system, it was found that although pass rates of the unit with the quiz system in place were similar to pass rates when the system was *not* in place, those who passed did 6% better on average in their final exam. It also became apparent that the feedback mechanisms in this quiz system were very effective - some students even liked to study the worked solutions of other randomised quizzes in detail before attempting their own assessed quiz.

Out of the several existing On-Line or Randomised Quizzes that have been developed, student responses have generally been favourable. Most evaluations of the quiz systems show that student exam marks increase when students use these assisted learning tools. This shows that online and randomised quizzes are a useful tool in automatically assessing students and helping them understand course content. The *Randomised Quiz System for Assisted Learning* aims to take the best aspects from all the quizzes mentioned in this section and create a new system for first-year programming students that is flexible, provides feedback and ultimately helps students achieve better results in programming courses.

3. Quiz System Design

This section of the report discusses the requirements that the *Randomised Quiz System for Assisted Learning* has, as well as the resulting designs that were derived from these requirements. Methodologies and design considerations are outlined, along with justifications for design decisions that were made.

3.1 Methodology Outline

When choosing an appropriate methodology for this system, several factors had to be taken into consideration. The most important of these being the lack of resources - both time (essentially limited to 9 weeks of development) and manpower (1 person). Taking these into account, a mix of agile and traditional programming methodologies seemed the most suitable for this project. The agile ideas proposed included delivering working code over extensive documentation, performing extensive testing and collaborating heavily with the project supervisor whilst developing the prototype. Consequently, almost no time was to be spent in analysis. There was to be a single design phase, and the coding was to be broken up into 3 phases - all delivering specific modules of the system.

When it came to actually fulfilling the design phase proposed, it became evident that the objectives for each coding phase were not sufficiently defined. Combined with a lack of familiarity with the technologies used in this project, it was decided that the singular design phase should be broken up into several smaller phases. This would also allow design phases to take into account what had already been accomplished.

The reason for breaking up the coding into three specific phases was to help modularise the system. Phase 1 concentrated primarily on XML Question design and implementation (outlined in *Section 4*); Phase 2 focused on extending Phase 1 and interfacing it with a compiler (outlined in *Section 4*), and Phase 3 focused on developing the “Front-End” part of the system (outlined in *Section 5*).

A detailed explanation of what modules were developed (and their interaction with the system) is outlined in *Section 3.3*.

3.2 Overview of Applicable Technologies

As mentioned in Section 2, other randomised quiz systems already exist and have used several different technologies in order to deliver the quiz to students. Some of the more successful quiz systems (specifically for Mathematics) use a combination of PDFs and Javascript for their quiz system

[8]. One of the main motivations for this is that Mathematical equations are easily displayed in LaTeX, which integrates nicely with the PDF file format. This would of course be unnecessary for this project as C/C++ uses plain ASCII text.

One advantage that PDF files have over other methods is the fact that the quiz gets downloaded once and the student may complete it offline. The quiz then gets marked and submitted later using embedded Javascript. Unfortunately, different vendors implement Javascript in PDFs differently, leading to unpredictability between platforms.

Other quiz systems have used basic HTML for their interface [3][5]. This obviously requires the student completing the quiz to be online at all times. It also means that the task of tracking the progress and correctness of the quiz attempt is dependent on a single server.

The main advantage of using HTML for the quiz delivery is that it is used almost universally. Every major Desktop Operating System has a web browser which supports HTML, Cookies, CSS and Javascript. Even smartphones and tablet devices now have a fully functioning browsers - meaning that students can complete their quizzes on a wide range of devices.

HTML is also advantageous because it can be outputted easily by most programming languages including PHP, ASP, .NET, Java and Python. For these reasons, a mix of current standard web technologies including XHTML, CSS and Javascript were used for the frontend of the Randomised Quiz System. Using these mature standard technologies will ensure that the system is compatible with as many platforms as possible.

3.2.1 Analysis of appropriate backend technology

The ‘backend’ technologies are technologies that the users of the system never come in direct contact with. For the Randomised Quiz System, technologies had to be considered for:

- Data Storage
- Program Logic
- Quiz “Engines”
- Question Templates

For data storage a MySQL database was selected. Since the quiz system would not require any complex database operations (triggers, stored procedures etc) and was not expected to have multiple concurrent users, a free open-source database was appropriate and less resource intensive than commercial DBMSs.

Since the front-end of the quiz system was to be HTML, PHP seemed a logical choice for the program logic. It was picked over other languages like ASP, .NET and Java because it is lightweight (although slower as it is interpreted and not compiled) and not dependent on proprietary technologies [14].

The quiz ‘engine’ is what either creates the questions, or validates them to make sure they’re correct. In some Mathematics Randomised Systems, the quiz engine is based on proprietary products like *Matlab* [2]. These products can help generate questions, validate answers and even give worked

solutions to questions. However in programming, worked solutions are less useful. Since this quiz system is testing C/C++ problems, it seemed logical to use a C++ compiler (GNU g++ or gcc in this case) for the quiz engine. The role of the compiler and a detailed explanation of the Question Templates is outlined further in *Section 4*.

3.3 System Structure Outline

This prototype was developed in two separate sections. The reason for this is twofold. Firstly, this helps adhere to the original specification of this project, which did not include marking mechanisms and reporting algorithms. The second reason is for the sake of expandability. If the backend is completely independent, there is the opportunity to implement frontends using other technologies. It also allows the system to be easily used in existing learning platforms.

The prototype consists of a “backend” and a “frontend”. Although the frontend contains significantly more functionality than solely presentation, it is referred to as the frontend throughout this document because it relies on the backend and does not implement any quiz generating functionality. A simplistic diagram of the components in both sections and how they interact is displayed below in *Figure 3.1.*

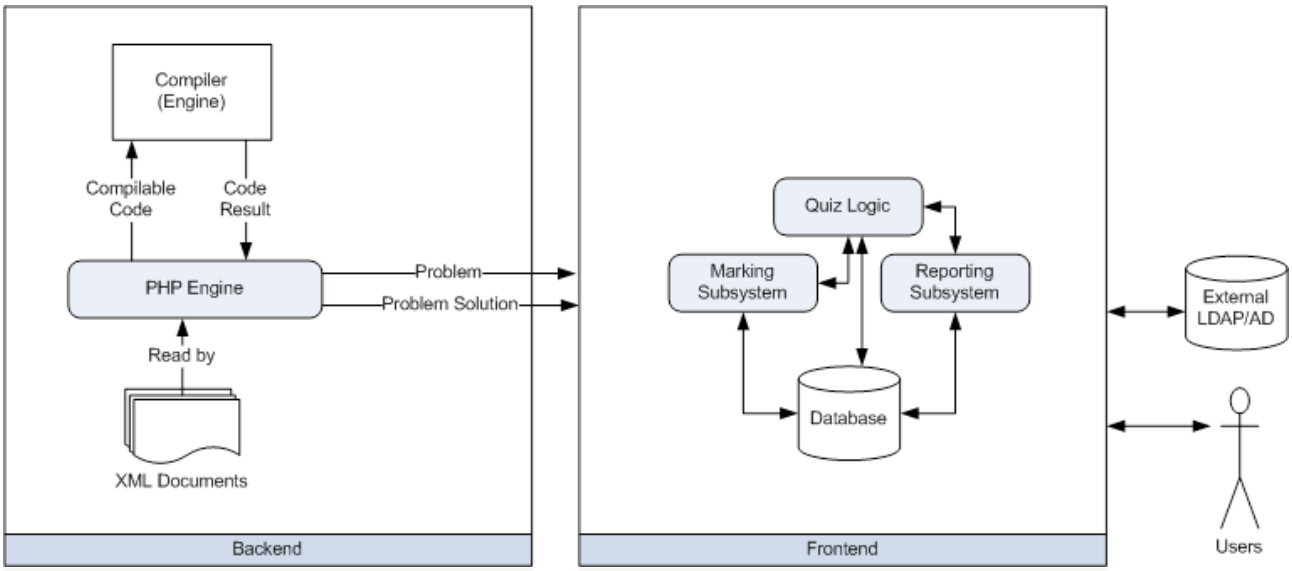


Figure 3.1 Component Interaction

The Backend essentially involves reading Question Templates (stored in an XML document), interpreting and randomising them using PHP, then obtaining the appropriate answers for the generated code using the compiler, and outputting the results. The details of this section and its components are outlined further in *Section 4*.

The Frontend contrasts with the backend significantly as it has nothing to do with generating random questions. It consists of components that assess quizzes (generated by the backend), generate reports, handle authentication and track student progress.The details of this section and its components are outlined further in *Section 5*.

3.4 Randomised Question Structure Outline

In order to understand what types of questions the *Randomised Quiz System* could support, an analysis of the types of questions that are common in introductory programming courses had to be performed.

In looking at both practical and examination material for introductory programming courses, it became clear that the questions structure for the Randomised Quiz System had to be extremely flexible in order to emulate the different types of questions found. In analysing these questions, it was found that several types of questions were common. These included:

- Questions that require students to write a solution to a problem detailed in text
- Questions that test comprehension of code by asking for output
- Questions that test understanding of code by using differing inputs and then asking for output
- Multiple choice questions that involve textual scenarios
- Multiple choice questions that test comprehension of code
- Questions that require students to fill in missing parts of the code based on a description or output
- Questions that ask for input where the code and output is given

Obviously not all these question types are appropriate for an automated quiz system.

3.4.1 Determining appropriate question types

Because the Randomised Quiz System is to generate questions (and corresponding instructions) from a template, it was decided early on that the system was not going to cater for questions that involve textual scenarios (as this would be hard to randomise) or questions that ask the student to write significant amounts of code.

The design that was decided upon allows three basic types of question to be generated:

- Code Comprehension Questions with students providing output
- Multiple-Choice Code Comprehension Questions where students choose the correct output
- Questions where students can fill in sections of the code to get a provided answer

Although these question types may seem limiting, they actually allow for a variety of questions to be asked given the randomised nature of the system.

It was decided that the ideal type of questions that should be generated from the quiz would be short and dedicated to one or two concepts. This would allow students to focus on learning one concept at a time and give educators a much more meaningful view of what concepts their students are having trouble with.

One issue that was considered during design of the Question Structure was the ability to provide feedback as to how a given answer was wrong. This has been achieved with multiple choice questions, and details of how this was accomplished is outlined in detail in *Section 4*.

3.5 Database and Concurrency Requirements

The randomised quiz system has significant database requirements. For this reason, detailed database design had to take place before development began. Database requirements were deconstructed into two main areas: Student requirements and Educator requirements.

Students require that their best mark for any quiz is saved. They also need to ensure that the quiz system saves their place in a quiz attempt, and that their entire quiz question history is saved in case they need to dispute a question.

Educators have significantly more requirements. They need to help classify what concepts each question tests, what concepts each quiz should test, how well students have performed in relation to each quiz or concept and the marks each student has received.

Taking these requirements into account, a relational database schema was developed. This schema is provided in *Appendix A*. This is the final schema used in the system.

As far as concurrency is concerned, this randomised quiz system would ideally be able to test around 300 students simultaneously. Using a standard web server to serve 300 pages simultaneously is not difficult [15], but unfortunately the quiz system would also have to *track* 300 students simultaneously. This would be achieved with the aid of a database, but would also rely heavily on session cookies - a feature built into PHP. For more information on concurrency, optimisation and performance testing, see *Section 5.10*.

3.5.1 Anticipated Performance Issues

Despite precautions taken outlined later in this report, there are still some issues that are to be expected when the system is put under load.

In the rare event that more than 100 users ask for a new quiz question at the same time (this is very different from there being 100 users *using* the system), delays of more than 80 seconds would be expected. Methods used to combat these delays are outlined in *Section 5.10 - Optimisations*.

Unfortunately, since the MySQL database doesn't support transactions like commercial DBMSs, there is the possibility that some of the data could suffer from concurrency issues. Precautions to prevent this were planned in the Design phases by ensuring referential integrity when necessary and ensuring the database design was simple enough so that ACID principles were easily enforced. The only problem I can currently foresee is 2 students could possibly get the same question if their request for a new question was intercepted at *exactly* the same time. This would be very rare as the entire database transaction of allocating a question to a student takes on average 0.0025 seconds.

Lastly, if many students log into the system (more than 500 at a time for example), PHP could have issues with dealing with so many concurrent sessions. This can generally be overcome by changing configuration files. Speed related issues resulting from an excess amount of users could also be resolved by increasing the memory available to the server running the quiz system [14].

3.6 Marking and Reporting Design Decisions

It was decided early on that the primary objective of the Randomised Quiz system was to help students understand concepts and to help them self-learn. Although the prototype is likely to be used as some form of assessment at some stage, it is mainly aimed at helping students test their programming knowledge.

For this reason, it was decided that when marking a student's answer to a question, there would be the opportunity for two attempts. The first "wrong" answer would not be taken into consideration when giving a final mark for the quiz; Instead, students would get another opportunity to answer correctly.

Although a numerical mark helps grade students, it was decided that timing mechanisms should be incorporated into the design. All quizzes will have a pass mark that is decided by the lecturer. In order to ensure that most concepts are actually understood, it is likely that a lecturer will set a high pass-mark (eg. 85%). With such high pass marks, it becomes difficult to rank students.

Therefore, it was decided that each question a student answers should be timed. These times will be compared to the average time taken for that question. Theoretically, from this information, a lecturer can see how well a student understands various concepts. It also provides a ranking mechanism that may be useful to lecturers.

From both the marks and time information collected in the database, there is the possibility to provide many different reports. While designing report functionality, it was decided (given the lack of time) that basic reports showing marks would be appropriate, along with reports that analysed the time taken on any individual quiz. A report showing statistics such as time taken on every individual *question* was also planned.

Although the number of reports planned was small, the database structure is flexible enough to allow many more reports to be generated, depending on what the educator desires.

3.7 Programming Practices

Since this project uses such a wide range of technologies including HTML, PHP, Javascript, C++ and XML, it was going to be necessary to adopt certain programming practices in order to keep the structure comprehensible.

In the front-end, Javascript files were to be separated by function. Since the main logic behind this prototype was to be written in PHP, there was going to be a substantial amount of PHP code that had to be managed. It would be necessary to keep track of what piece of code provided what functionality. It was decided that 'separation of concern' needed to be implemented into the code to provide a clear structure and easy maintainability. The method used to achieve this was to loosely follow the Model-View-Controller (MVC) programming practice. Most files that the user would view would come under the "view" category, while the functions that these 'view' pages would call would be stored in other files that are essentially the "controller/model" pages.

Each PHP file that is in the prototype is clearly named help to understand the function it achieves. The code is documented, and each file is generally limited to one task only. For example

content_quiz_add.php handles the form that is displayed when an educator wishes to add a quiz. The *content_quiz_add_process.php* page processes the data entered in the form and performs the appropriate error checking needed for the form etc. Both files depend on the *class.Quiz.php* file which contains functions that relate to Quiz reading/updating/modifying quiz objects.

It was also decided that this project should use Object-oriented practices whenever possible. Each major 'object' in the system was to be closely correlated with corresponding tables outlined in the database. This would make the code easier to read and understand.

3.8 Design Summary

Although the concept of a randomised quiz system seems simple, there are many technologies involved which need to communicate. Careful orchestration and design is needed to ensure that these technologies interact and communicate with each other sufficiently.

The design phases of this quiz system involved breaking up the system into two parts - frontend and backend - then defining each high-level component that was to be used in the system and how it was to interact with other components in order to perform the required tasks. From this point, design diagrams were manufactured that depict how certain operations within the prototype were to be accomplished. Since this report is not a design document, these have not been included.

Separating the backend and frontend will allow future applications to utilise the quiz generating core of this prototype. Adopting an MVC-like approach to programming and providing a flexible database design also meant that the entire prototype would be flexible, easily maintainable and able to be extended with minimal effort.

4. Back-end Implementation

As discussed previously in *Section 3 - Design*, the Randomised Quiz System for Assisted Learning was to be developed in two separate parts. The implementation of the backend is outlined in this section. This essentially is the *core functionality* of the quiz system and manages the reading and interpreting of XML Question templates, and the generation of randomised questions with the aid of a compiler.

The main components of this part of the system are:

- The XML Question Templates
- The Quiz Logic
- The Compiler

Each component and its implementation is detailed below.

4.1 Question Templates

At the heart of the randomised quiz system are questions. The randomised quiz system must generate questions that are C++ problems that compile and make sense to a student. It must also provide appropriate feedback when necessary.

Although the questions that this system outputs need to be random, they are more than likely going to follow a specific pattern or template. Take for example the question outlined in *Figure 4.1*.

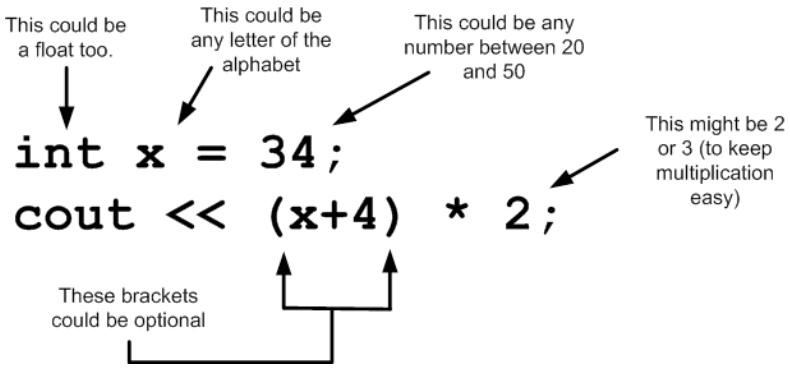


Figure 4.1 Determining random aspects of a question

Figure 4.1 shows a very simple question testing basic arithmetic. It has many parts that could be randomised. If these parts were to be randomised, the same single question template could have several hundred different problems generated from it.

Because the question template structure was to be human-readable, machine-readable and as flexible as possible, XML was chosen as the implementation language. Each XML document follows a certain schema to ensure that it can be interpreted properly by the system.

Each XML file has a basic ‘template’ problem inside, along with tags that help classify it in within the system. These tags include difficulty, concepts tested and the time that the question is expected to

take. Without the 'random' component, a skeleton of an XML question template look like what is shown below.

```
<question type="output">
	<estimated_time>180</esimated_time>
	<concepts>
		<concept>Incrementation</concept>
	</concepts>
	<difficulty>1</difficulty>
	<instructions>What is the given output of this program?</instructions>
	<problem>
		#include &lt;iostream&gt;
		using namespace std;
		int main()
		...
	</problem>
</question>
```

The question type outlined corresponds to one of three that can be handled by the system: output, fill-in or multiple (see *Section 3.4.1* for why these three were chosen). Each question template can assess multiple concepts and has a difficulty associated with it. The C++ problem is simply written within the `<problem>` tag. This provides a logical and flexible question structure.

4.1.1 Using Substitutions

As shown in *Figure 4.1*, there are many parts of a question that can be randomised. These parts are called 'substitutions' in this prototype. PHP is used to populate these substitutions. An example of a complete XML question template is outlined in *Appendix C.* Essentially each substitution corresponds to a section of PHP code. Then, in the original `<problem>` tag, the insertion of the substitution is achieved by enclosing the substitution identifier in " ` " characters.

For example, if we take the C++ code:

```
int x = 24;
```

But we wish to substitute 24 with a random number between 20 and 25, the problem code would be:

```
int x = `s1`;
```

Where s1 is the identifier for the first substitution. It's corresponding XML might be:

```
<substitution val="s1">rand(20,25)</substitution>
```

The `rand()` function is built into PHP. However, the functions used here are not limited to built-in functions. User-defined functions can also be used. It is also possible to refer to previously calculated substitutions by enclosing their identifiers with %'s. Eg. `%s1%-1` would return `23` if the `s1` substitution equalled `24`.

PHP is used in these substitutions for two reasons. Primarily, it's the easiest to code as the quiz logic is written in PHP. Secondly, it allows for a great deal of flexibility - allowing loops, conditional statements and custom-written functions.

4.1.2 Alternate Answers

Since the quiz system allows multiple choice questions, there had to be a way to make the wrong multiple choice answers appear feasible. This becomes difficult as the question content (and therefore

the answer) is randomised; therefore the 'wrong' answers in a multiple choice quiz should also be randomised in some way.

In order to implement these 'alternate' (wrong) answers, it was decided that substitution and recompilation of the problem code was needed. The way this works is that the quiz system changes key (specified) sections of the question code to make the program behave differently. It then recompiles the question source code and reads the output the program gives when executed. This output then becomes an alternate/wrong answer. A data flow diagram depicting this process can be viewed in *Appendix B*.

If we take for example the source code below:

```
int x = 2;
if (x=3){
      for(int i=0; i<2; i++){
            cout << "Hi ";
      }
}else{
      cout << "Bye";
}
```

You can see that this program is testing both boolean operators and loop structures. The correct output for this program is `Hi Hi`, but it has a couple of misleading statements that a first-year programming student might not understand. The first is an assignment operator used as a boolean condition; ie. The operation `x=3` will always be successful in this case, and therefore will be `true`. A novice might read this as `x==3`, a condition that would change the output of the program to `Bye`.

The question author might put this in as a common error, and simply substitute = with == in one of the alternate answers. The author also has the opportunity to write a description of *why* this answer is wrong. This works in conjunction with other substitutions specified in the original question (eg. The 3 might not always be a 3). Other alternate/wrong answers might involve changing the `i<2` to `i<=2` to see if the student understands how loops and comparison operators work. Again, this would change the output and the student would be presented with the feasible answer `Hi Hi Hi`.

The reason that substitution and recompilation is used is so that students get multiple choice answers that appear to be feasible. This mechanism also gives the question authors the chance to give the student feedback on why their answer was wrong. This helps encourage self-learning and still keeps the quiz randomised.

4.1.3 Fill-in answers

The other question type that was implemented was a "fill-in" question where students had to implement a partial solution to a question that was given.

Again, this used a method of substitution where the 'correct' answer is defined by the question author, but is marked as section of code that is not to be displayed. A diagram of how this is implemented is shown in the figure below.

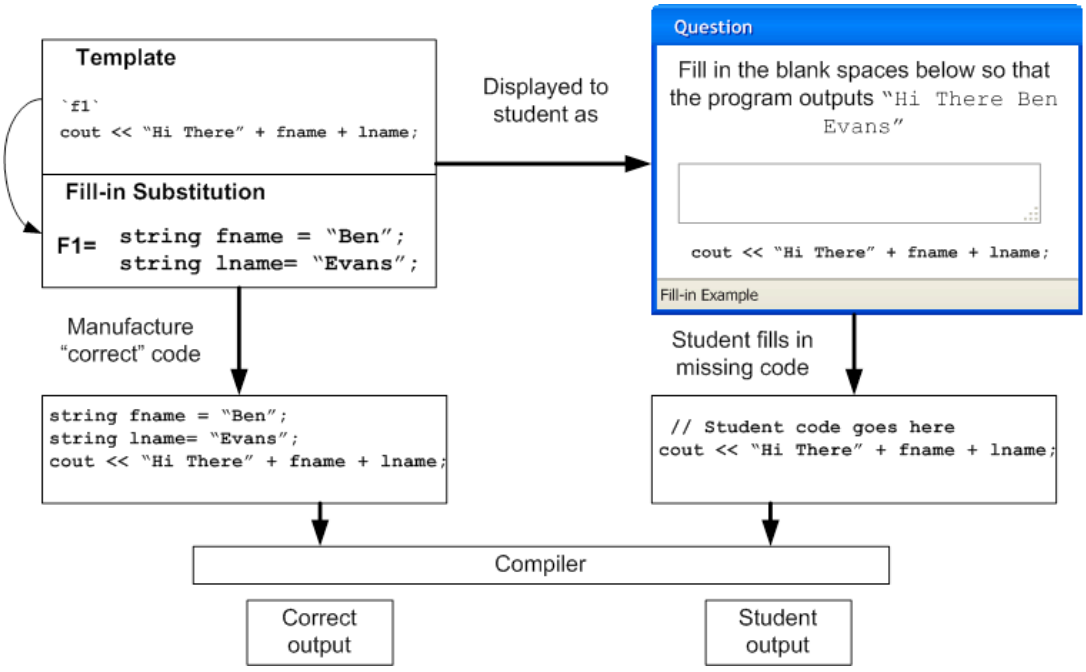


Figure 4.2 High Level view of "Fill-in" Questions

The good thing about this method of implementation is that the author can specify their implementation of the solution, and the student can specify their own method of obtaining the correct result. As long as both solutions manufacture the same output, the student's answer is marked as "correct". This allows for a great deal of flexibility, but there are security concerns with this method and the possibility of students being able to cheat. This is discussed further in *Section 6*.

4.1.4 XML Structure Flexibility

This XML structure allows for a lot of flexibility. It enables question authors to specify exactly what parts of the template questions are to be randomised using PHP code, and allows the system to generate feasible multiple choice questions while still keeping the quiz random. This flexibility theoretically allows more difficult concepts to be taught - eg. Pointers. Since the structure is relatively simple and not really language dependent, it could be used with other languages like Java and Python. These possibilities are outlined further in *Section 6*.

4.2 Interpreting XML Questions

The next step in implementing the backend was to create the quiz logic subsystem that could interpret the XML question templates and output a randomised piece of code and its corresponding answer.

The actual processing of the XML file was done using a freely available XML parser for PHP. This works mostly on the XML DOM and puts the entire document into a series of arrays. From here, various parts of the XML DOM were directly put into object variables.

The most difficult part of the quiz logic was correctly implementing the substitution functionality. As mentioned in *Section 4.1*, each substitution in a template file is written in XML. PHPs `eval()` function was used to evaluate the code that was parsed from the XML file.

After the substitutions were performed, the quiz logic would substitute the appropriate variables in the "multiple choice" sections (details of this can be found in *Section 4.1.2*), and remove the appropriate code for the "fill-in" questions. After this had been performed, a fully compilable 'question' would be created.

4.3 Use of Compilers

The compiler is essentially the *engine* of the Randomised Quiz System. *Section 4.1* has outlined how parameterised substitutions were used to generate a piece of code. However, if the output to this code is not available then the question cannot be marked. In order to find the value of the code, a compiler is used.

Once the fully compilable question is created, the quiz logic subsystem writes this code to a file in a temporary directory on the server's hard drive. The logic subsystem then calls the compiler to compile the code and outputs the executable to another similarly named file. From here, the logic subsystem instructs the operating system to run the executable that was created and put the output of the execution to a file. This output file is then read by the system and designated the quiz "result". A high-level view of this entire process is outlined in the Figure below.

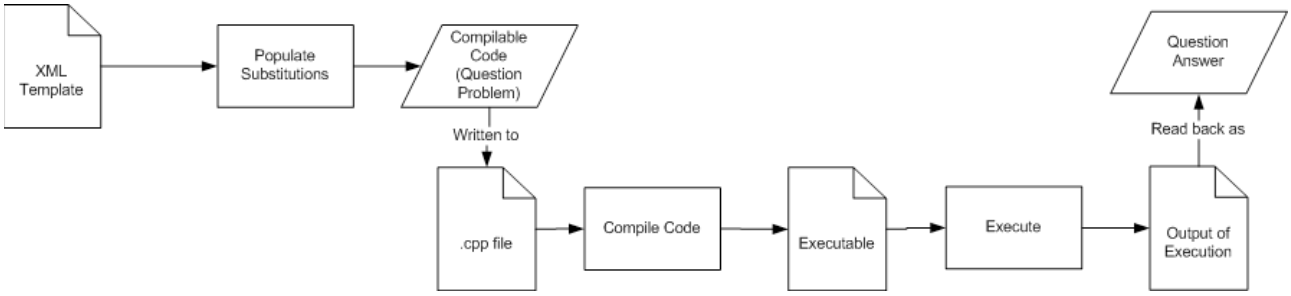


Figure 4.3 Data Flow showing how answers are obtained

This method contrasts to other implementations outlined in *Section 2*. The code is not interpreted, but compiled. Also, the compiler is an engine, but it is not used to generate the code - that is handled by the logic subsystem. The compiler in this prototype is used only to validate that the code generated by the logic subsystem and to compile that code so that its output may be read. Unfortunately, since this method relies on the standard output of the operating system, there must be some output function in the original code. In C++, the output is usually achieved using the `cout` keyword.

4.3.1 Dealing with Troublesome Code

The compiler is also used in this system to validate student answers when they have to fill-in code. For example, they might be completing a loop function. Unfortunately, there is the possibility that this loop might be infinite. If the operating system runs this compiled program, the program will run forever unless manually stopped.

Although this sometimes isn't intentional, this type of code is handled as malicious code because it could lead to a denial of service. It was decided that this denial of service could be avoided if each program was allowed to run only for a specific amount of time.

Initially it was thought that PHP could do this with it's built in functions. However, it was discovered that PHP would just resume its script execution while leaving the process it was waiting for still running. It became clear that this had to be done in the operating system instead. In order to achieve this in Linux, the keyword `timeout` was used. In windows, this operation was significantly more difficult. To limit execution times, a program was written in scripting language *AutoIT* that spawned a process, kept track of its execution time and forcibly 'killed' it if it didn't terminate.

Although this solved the problem of long-running programs, it didn't solve the problem of programs that maliciously try and infiltrate or cripple the server. Potential solutions to this are outlined in *Section 6*.

4.4 Environment and Technologies Used

As mentioned in *Section 3*, the technologies selected in the design phase of this project are supposed to be completely open-source and cross-platform. While developing this prototype, it was tested in two completely different environments. The basic requirements for this prototype to run are:

- Web Server - enabled with PHP (>5.0)
- GD2 Capabilities for PHP
- LDAP Modules for PHP
- MySQL Webserver (>5.0)
- A C++ Compiler
- Linux, Unix, BSD or Windows

The platforms tested were Linux (*Ubuntu Server* - v10.10 x86) and *Windows Server 2003* x86. Both operating systems used the most up to date service packs and latest stable versions of the software available. On both platforms, the latest precompiled *Apache Webserver* (v2.2.17) was used, along with the latest version of *PHP* (v5.3.3) with LDAP support included. The compiler used on both operating systems was the *GNU Compiler Collection* (gcc and g++).

Overall, both systems behaved similarly. The only operating system specific code that had to be written was the code that limited execution time (See *Section 4.3.1*) and code that used antialiasing in images. Excluding this, all other code runs identically on both operating systems, making this prototype truly cross-platform.

4.5 Backend Implementation Summary

The backend implementation of this Randomised Quiz Prototype provides a very flexible, innovative and powerful XML question structure that is capable of generating very simple and very complex randomised questions for students to answer.

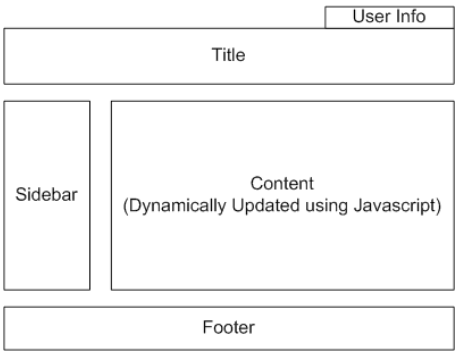
Using parameterised substitutions (implemented in PHP) could theoretically mean that thousands of unique questions could be generated from a single question template. The backend is written in such a way that any frontend can interface to it with minimal difficulty. It is also cross-platform and extensible.

5. Front-end implementation

This section outlines the way that the remainder of the quiz system has been implemented. It explains various techniques used to render graphical user interfaces, accept and mark student responses, recover from errors, cache data and authenticate from external sources. This section also provides a high level overview of how the front-end components function separately and explains how they were implemented.

5.1 User Interface Outline

The user interface for the front end part of the Randomised Quiz prototype is in HTML. This is a logical choice considering the program is implemented in PHP. The user interface uses a standard website layout with a sidebar, title and content area. This user interface 'shell' is used throughout the frontend.



In order to reduce the load put on the server, the user interface 'shell' is only loaded once. From here, each new page or section is loaded into the content area using an asynchronous Javascript request (AJAX). This has several advantages. The most important advantage is that every page loaded into the content area only has to worry about its own functionality, and not worry about various presentation elements. This separation of concern is also good programming practice. Using AJAX also means that the client is responsible for updating the content area instead of the server (this would mean using messy HTTP GET variables to tell the server what page is to be displayed).

5.2 Creating Quizzes

In the frontend, quizzes are created by educators (who have the appropriate privileges). In order to create quizzes, it was decided that the educator would firstly provide basic quiz information such as the date opened, the deadline, the name etc. It was then decided that they should provide information as to what concepts should be tested, and at what difficulty these concepts should be.

Since each XML question template (*Section 4*) must provide a difficulty level and a set of concepts that the question tests, this was relatively easy to implement. Structuring the quiz this way also ensures that students will not always get the same question 'template'.

The concepts and difficulties being assessed in this quiz are listed below:

NUM OF QUESTIONS	CONCEPT	DIFFICULTY (FROM)	DIFFICULTY (TO)	ACTIONS
10	Arithmetic Operators	1	1	✕
10	Precedence Rules	1	2	✕
10	Incrementation	1	3	✕

Figure 5.1 Educator Interface for Adding Questions to a Quiz

5.3 Intelligent Question Selection

There has been an attempt to personalise the quiz to cater to individual students. This is performed by implementing algorithms that choose questions for students.

When taking quizzes, the system must fulfil the question requirements specified in the quiz outline - eg. 10 Questions on the concept “Arithmetic Operators” must be completed with difficulties ranging from 1 to 3. This would suggest that roughly 33% of the questions on Arithmetic Operators asked would be of difficulty 1, 33% would be of difficulty 2 etc.

In order to make the quiz more personalised, algorithms were introduced that take into account how well the student is doing in relation to a concept, and how long they’re taking on average. If a student is struggling with a concept (which can be signalled by taking more than the average time or consistently getting questions wrong), the system will give them more of the easier difficulties to help them understand the concepts better. Conversely, if the student is correctly completing the questions in times that are below the average, the quiz system will give them harder questions to help challenge them more. This not only gives the quiz a personalised feel, but also helps ensure that each student understands the concepts tested in the quiz.

The way this algorithm was implemented was simple. The number of questions to be asked at each difficulty is always evenly distributed (eg. 3 easy, 3 medium, 3 hard). What was introduced was a buffer of 20%. The buffer caters for each individual student. Eg. The buffer will most likely consist of easier questions if the student is struggling with a concept. It will consist of harder questions if the student is doing well.

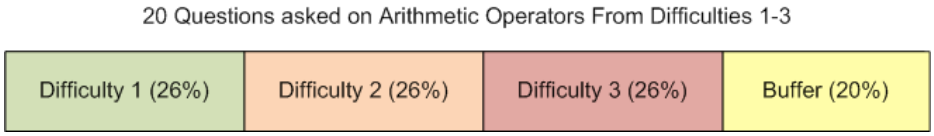


Figure 5.2 Intelligent Question Selection for 3 difficulties.

The algorithm is also progressive so that if a student initially does well, and then struggles on a ‘medium’ difficulty, the buffer will be filled with medium difficulty questions. If a student struggles a little with each difficulty, the buffer will consist of questions from all difficulty levels.

This algorithm will help students understand the concepts involved in each quiz while personalising the education experience.

5.4 Marking Subsystem

The marking subsystem was a simple yet critical part implemented in the Randomised Quiz System. Whenever students submitted answers, generally the output that they provided had to be checked against whatever the program’s output was.

As mentioned in *Section 3*, it was decided that the marking subsystem would allow students to be given a second chance for every question they got wrong. The marking subsystem puts a number of attempts for each question in the database but awards the student full marks if they got the question right on the second attempt.

Generally the marking algorithm is simple. String comparison operators are used in PHP to see how similar the program output is to the student's response. If the two are the same, then marks are rewarded. However, with "fill-in" questions where students have to write code, a more complex algorithm is used. The student's code and the given code are merged to create a compilable C++ file. The file is then compiled and the code output is compared to the author's code output. If they are the same, full marks are rewarded. *Figure 4.2* outlines this process.

5.4.1 Ensuring Marking is "Fair'

When testing the system, it became apparent that PHP, C++, Linux and Windows all handle newlines in different ways. PHP, C++ and Linux appeared to use one convention, while Windows (and form input) appeared to use another. For example, if the code `cout << "Hi" << endl;` was used, and the student outputted:

```
Hi
<newline>
```

The string comparison operator would fail. This was because C++ program used newlines and carriage returns, while PHP interpreted the student's answer as only having a newline.

To combat this, while comparing strings, all newlines (both conventions) are completely removed and then compared. This unfortunately means that the concept of newlines cannot be examined by the Quiz System, but it ensures fairness when marking.

5.5 Plagiarism and Cheating Prevention

Since the Randomised Quiz System prototype is online and written in HTML, several problems in regard to cheating present themselves.

Firstly, most of the questions in the quiz system involve students reading what a certain piece of code does, and then giving the corresponding output. If the problem was given in plain text it could be very easy to copy and paste the problem into a compiler, run the program and find the output. This was thought of early on in the implementation. To combat this, all comprehension problems are dynamically put into an image using the GD2 library for PHP. Essentially this means that a picture is displayed (which has text inside it). Since students cannot easily get the textual content of the picture, they cannot copy and paste this into a compiler.

Unfortunately, other features that would be easy to implement in plain text like syntax highlighting had to be excluded from the prototype as they were too difficult to implement in an image.

Another foreseen problem was the manual entry of the code into a compiler. Since students are not able to copy the text of the problem (because it is an image), there is the possibility that they might transcribe the text into a compiler. To combat this, each question template has a maximum time limit. Entering text manually generally takes more time than attempting to comprehend a problem. If the student exceeds this recommended time limit, this can be seen by the educator. If students regularly exceed recommended time limits for questions, educators can deduct that they are not understanding the questions or attempting to cheat.

5.6 Coping with Errors

Unfortunately during testing, the backend part of the quiz system did not always function as intended. On occasion, a question would be generated that wasn't compilable, or contained a logic error (like an infinite loop). Students shouldn't be expected to answer malformed questions; consequently the need to handle errors elegantly was needed for the front-end.

Before allocating a question to the student, the frontend makes sure that:

- The backend delivers a question that is not blank
- The backend delivers a corresponding solution that is not blank
- The backend delivers 3 wrong multiple choice answers (if the question is multiple choice)
- If multiple choice answers are given, there should be no duplicate answers

If the backend does not deliver what is expected, the frontend will disregard that question once and ask for another. If the backend still delivers a corrupt question, the frontend will choose another similar question (if available) that has the same (or lower) difficulty and tests the same concept(s). Finally, if there is no other question available or the new question is still corrupt, the program will give an error. This ensures that errors in the Question Author's templates that haven't been correctly tested will not affect student questions. Every question displayed to students is solvable.

Another issue the frontend deals with is student errors. With "fill-in" questions, students have the opportunity to write code. This code could contain malicious instructions (discussed more in *Section 6*) or could result in an infinite loop. Since this code is compiled and run on the Quiz server, this is a concern. Like badly written questions (discussed in *Section 4.3.1*), student code could result in excess server resources being allocated and consequently denial of service. To resolve this, the frontend applies the same algorithms discussed in *Section 4.3.1* to ensure the code terminates within a reasonable amount of time.

5.7 Integration with Database

The database outlined in *Section 3* was not actually utilised during the backend development as it is not needed to generate questions. It is however needed to perform nearly all frontend operations.

The frontend of the Randomised Quiz System is object oriented. These objects correspond very closely to the database. Each table in the database is effectively an object in the frontend. Each quiz that is specified by an educator is stored in the database. Each student's attempt at a quiz is stored. Each question attempt is also stored. However, the question templates are never stored - their attributes such as estimated_time, difficulty and concepts tested are cached though.

In order for an XML question template to be classified as 'valid' and consequently cached, the frontend firstly tests it to make sure that there are no syntax errors (eg. Putting a < instead of < in the question problem). It then ensures it can parse all necessary cached information properly (difficulty, estimated time etc) and checks to make sure it's not blank. Lastly, a test question and its corresponding answer is generated. If everything is successful, the XML question data is cached and designated a 'question template'. Part of the reason for this caching of the XML data is for speed purposes (eg. Not having to scan all XML files to find out what difficulty each has), but it also helps when referential integrity has to be ensured.

5.8 Integration with AD/LDAP

Since the Randomised Quiz System is designed as a standalone system, a form of authentication that used existing student credentials was desired. To accommodate this requirement, integration with Active Directory and/or LDAP was implemented in the prototype.

Active Directory or LDAP provide a means for the quiz system to externally authenticate all users. External authentication was chosen over internal authentication for the following reasons:

- The external directory will already securely store student and staff passwords
- There is one less username and password for students and staff to remember
- The external directory will have up-to-date class lists
- The external directory already has appropriate groups set up
- If the Randomised Quiz System is somehow compromised, no credentials will be exposed

The external directory service doesn't only provide authentication. It also plays a role in assigning students to quizzes. When an educator 'opens' a quiz, they define who this quiz applies to. If for example there are 50 quizzes spread across 5 classes, it is undesirable to have students accessing quizzes that aren't intended for them. The directory service allows users to be assigned into *groups* [15] that have certain permissions. The educator simply has to specify what directory group a class of students belongs to. Most IT managers assign groups to students based on the classes that they are enrolled in, thus making restricting quizzes to particular classes easy.

The external directory service also plays a significant role in reporting and marking algorithms. Each quiz attempt is tied to a username that is present in the directory service. This username by itself may provide no obvious identification - eg. *S4072905*. The directory service allows the Randomised Quiz System to easily query a username and retrieve information about that user, such as their first and last name, date of birth, department etc. Thus eliminating the need to provide a username-to-name mapping. This allows for much more useful and detailed reports. The information in the external directory is also more likely to be updated than information in an internal independent database.

The actual implementation of Active Directory authentication into the Randomised Quiz system was done with the aid of an open-source library called PHP ADLDAP [17]. It supports Active Directory and numerous LDAP directory services. Upon implementation, it was found that querying Active Directory for large amounts of student data was slow. Therefore there is a local cache of users and names in the database to help speed up marking and reporting operations.

5.9 Report Implementation

The database in the Randomised Quiz System contains data about what quizzes were attempted, how long the attempts lasted, which questions were attempted by whom and numerous other pieces of information. Because of the amount of data stored, there is the potential for numerous reports to be manufactured.

The first and most essential reports implemented in the prototype involved students and their marks. Educators obviously want to see what students have passed their quizzes. The pass/fail report was easy to implement because of the methods written for the Quiz and QuizAttempt classes. The report

firstly queries Active Directory to find all members of the selected authentication group (usually a class list as discussed in *Section 5.8*). Once all members are retrieved, they are sorted by their last name. Their highest-rated attempt is then fetched using `QuizAttempt::getHighestMarkQuiz(user, quiz)` method. Similar methods are implemented in most objects in the system to make future reports easier.

Another report implemented used the PHP GD2 image library and the *JPGraph library* [18] to help create graphs that represent the time taken for each quiz. This report was implemented so educators could get an indication of how hard each quiz was for students. Another report implemented gives a complete summary of a given question. It shows the average amount of time taken for that question, what percentage of students get it right first time, a summary of its prescribed difficulty and a sample problem generated from it.

Although only three reports were implemented in the prototype, the data stored in the database and the methods implemented in the PHP code allows many more creative and useful reports to be generated.

5.10 Optimisations

Under ideal conditions, the Randomised Quiz System prototype functions quickly and with minimal errors. Several tests were performed on the system using *Apache JMeter* to see how the system coped under light use, and under stress. The tests were conducted 10 times with a simulated load of 5 and 100 simultaneous users. The table below gives a summary of these tests. All units are in milliseconds.

Test Type	Win Avg.	Linux Avg	Win Median	Linux Median
Basic PHP with Session Cookie	174	121	58	56
PHP page with limited database queries	84	102	59	56
PHP page with many database queries	782	765	781	767
Generating (and compiling) a new question	5201	3783	5205	3733

Table 5.1 “Light” load testing - 5 Simultaneous Users

Test Type	Win Avg.	Linux Avg	Win Median	Linux Median
Basic PHP with Session Cookie	1768	1120	1349	989
PHP page with limited database queries	1973	1455	1409	1397
PHP page with many database queries	3321	2982	2015	2450
Generating (and compiling) a new question	92400	79685	93657	81510

Table 5.2 “Heavy” load testing - 100 Simultaneous Users

As you can see, normal database operations are quite efficient, but generating quizzes (with 100 simultaneous users) takes a long time - especially on the Windows Server 2003 machine (Over 90 seconds). Because of this, optimisations needed to be put in place. It should be pointed out that although these times are *very* high, having 100 people ask for a new quiz solution at *exactly the same time* is very unlikely. Also, the machines tested (A Windows Server 2003 Pentium 4 3.8ghz; 512MB RAM, and an Ubuntu Linux Virtual Machine with 1 core and 2GB of RAM) are fairly underpowered.

Regardless, a solution to these excessive wait times was to generate the randomised questions in batches.

The delivery of a randomised question involves compiling. When analysing machine statistics, it was found that compiling C++ programs was particularly CPU intensive. It was also found to be the slowest part of the randomised question generation. To attempt to combat this, a PHP script that instructed the backend to generate 1000 questions was written. The script ensured that the questions and answers were generated, and the result was put in the database. This way, a simple database query (usually less than half a second, and an average of 1.5 seconds under a heavy load) is all that's required when a new randomised question is requested. This script could possibly be run in either a Cron Job (Linux) or a Scheduled Task (Windows) at times where the server is under minimal load (eg. 2:00AM).

Another optimisation already mentioned in *Section 5.8* is the caching of directory service information. Since querying external directory services is slow, active directory usernames and corresponding first and last names are cached in the database. This information can be accessed quickly. The information is also current - each time a user logs into the system, their first and last name is updated in the database. All user information can also be manually 'synced' with Active Directory if desired.

The final optimisation in the Randomised Quiz System is in the database. Since most queries in reports (and in general application use) are based on usernames, several indexes have been added in the database to help speed up operations. Although the performance benefits currently don't show, they will become apparent when the database grows.

6. Results and Future Work

This section outlines what was achieved in undertaking this project, as well as the known limitations of the finished product, and possible future extensions that could be implemented.

6.1 Initial Expectations vs Delivered Prototype

The initial outcomes for this project were to design and develop software that could generate randomised quiz questions for C++ aimed at first year computing students. In doing so, an understanding of what types of questions that would be suitable for this system should be developed.

The initial expectation of this project was to develop a prototype that read question templates and outputted a randomised question and associated answer. Initial expectations were that this sort of question could then be imported into existing Learning Management Systems like Blackboard or Moodle. After several weeks working on the prototype and collaboration with the project supervisor, it became clear that several extensions to the original specification could be added. The new aim was to deliver a system that could actually be used by students and staff at Macquarie University and implemented in 2011.

This project has delivered a prototype that not only generates questions and associated answers, but also a fully interactive quiz system that assesses, tracks and gives feedback to students. It also has a

comprehensive set of functions for educators to help develop quizzes and assess individual students' progress. With this quiz system prototype, there have also been 40 template questions delivered which demonstrate the flexibility of the question structure and showcase suitable questions. Question suitability is also outlined in various sections of this report. An additional report outlining details and syntax of the XML Question template structure (with associated examples) has also been delivered.

6.2 Appropriate Examinable Questions

In developing the Randomised Quiz System for Assisted Learning, several rules had to be developed in relation to question types that could be examinable. The following question types are possible in the Randomised Quiz System:

- Comprehension (looking at code and giving its output)
- Multiple Choice (Similar to comprehension, but with several selectable answers)
- Fill-in-the-blank (where parts of code are given, and students must fill in other parts)

The main requirements of any question that is examined is:

- The question must be fully compilable. This may mean including namespace and a `main()` function.
- The question must end in a reasonable amount of time (no infinite loops for example)
- The question must output to console. This is usually achieved using `cout` in C++.
- The question may not use any input from the console (no `cin` in C++).

This narrows down the types of questions that can be asked considerably. It does however still allow for a lot of flexibility.

Several first-year programming concepts have been tested in this prototype such as Arrays, Basic Arithmetic Operations, Functions, Boolean operators, Queues, Stacks, Loops and pointers. All these concepts are easily tested using any of the three question types.

It was discovered that using 'Fill-in-the-blank' question types was problematic in some instances unless the final output of the program was already written. This was mainly due to students misreading the instructions for that question.

In summary, the amount of suitable questions that can be asked in this quiz system is vast. Any question that doesn't violate any of the rules above is theoretically examinable. In practice however, it was found that testing many shorter programs that assessed one or two concepts was better (and easier to debug) than longer programs that assessed several concepts. Essentially, the suitable types of questions for this prototype is limited only by the Question author's imagination..

6.3 Limitations

Although the prototype for this project is comprehensive, there are some known limitations.

Firstly, all question templates must have an output to the console, otherwise they will not be able to be marked. This is due to the way that answers are obtained and compared. Programs cannot have console input either.

Another limitation is the lack of input checking from “fill-in” questions. Currently, C++ code that students write is compiled and executed on the server that hosts the Randomised Quiz System. This only happens in fill-in questions, and generally the user in which this program runs has limited privileges in relation to the filesystem etc. It does however pose a security risk. A solution to this is outlined in *Section 6.4*.

6.4 Possible Extensions

Although the Randomised Quiz System is usable and could sufficiently assess several classes in its current state, there are several possibilities for extensions.

The first possible extension could involve transplanting the backend of the prototype and using it to integrate with other Learning Management Systems. Since the *Randomised Quiz System for Assisted Learning* has two separate sections (a backend which generates the questions and a frontend that handles marking, authentication etc), it is possible to isolate the question-generating backend from the remainder of the prototype and run it independently. This essentially means that a batch of questions could be generated and output to an XML file and imported into a Learning Management System such as Moodle or WebCT/Blackboard. This may be preferable in some cases for the sake of consistency and integration, even though some features such as personalised feedback may be lost.

Following upon the idea of isolating the backend, one could also use the quiz system to generate mass amounts of questions and output them to PDFs so that personalised exams or written quizzes could be generated. A similar attempt at this for a Randomised Quiz in Mathematics is outlined in [3]. Another extension might be re-writing the frontend in another language - eg. Flash (for extra interactivity).

Since the prototype developed doesn’t rely on any specific C++ code, it would be relatively simple to allow the Quiz System to generate questions in other programming languages. All that would be needed would be another XML tag that signified the programming language that each question template was written in. Question structure and substitution could remain the same, regardless of the programming language assessed. The only requirement is that the programming language selected has a compiler that works on Linux & Windows and that outputs to the console. Possible languages include Java, Python, Pascal and even web languages like PHP.

Another possible extension could be resolving the shortcomings outlined in *Section 6.3*. For example, an algorithm that screens input that comes from the student so that no malicious code is ever run on the server. This might be done by scanning for keywords in the code (eg. Cout, return etc.), limiting the number of characters and optionally comparing student code to the author’s response. An algorithm for this was actually developed, but there was insufficient time to implement it properly for the prototype.

Another extension could be writing a user interface that helps generate XML question templates. Although the template format is very simple, having an easy-to-use interface would allow educators with less XML experience to be able to create new question templates with ease.

Another feasible extension could be the implementation of more reports. This prototype had only three main reports implemented, but the data stored and the objects and methods created in the quiz logic subsystem should allow many other reports to be created with ease.

Lastly, there is the opportunity to extend the base functionality of the Quiz System so that it can assess different types of questions. The XML structure in its current form is flexible enough to allow these changes, and the corresponding PHP code is well documented so as to make such an extension relatively easy.

6.5 Results Summary

The prototype developed for the *Randomised Quiz System for Assisted Learning* is comprehensive and meets the original specification of providing a system that can generate randomised questions for programming. Although there are several small limitations, the prototype is functional and flexible. There are also several extensions applicable to the system that could further expand its potential.

In order for the quiz system to be implemented in 2011, several more course-specific programming templates would have to be made, and a dedicated server running Linux at Macquarie University would have to be provided. It is recommended that before any 'fill-in' questions are provided, an extension preventing all malicious code from running (outlined in *Section 6.4*) should be implemented. When this is complete, the system will be ready to be deployed.

7. Conclusion

In conclusion, there are many motivations behind creating systems such as the *Randomised Quiz System for Assisted Learning*. Perhaps the most important of these is allowing students to receive instant and personalised feedback on work completed and allowing them to practice examples and questions to help boost final examination scores.

Although several randomised quiz systems have been previously developed, the *Randomised Quiz System for Assisted Learning* is a system that combines the best aspects of all previously developed quiz systems to provide the best quiz experience possible for programming students.

The prototype developed was implemented in two separate sections to allow for maximum flexibility and extensibility. The prototype allows randomised questions to be generated from XML template files. It allows educators to easily specify quizzes and also allows students to authenticate into the system and attempt these quizzes in their own time. The marking system is comprehensive and the entire system uses the latest standard web technologies. There are several optimisations in place to ensure the quiz system works well under stress and there is still the opportunity for future works to be developed to extend this quiz system further.

The *Randomised Quiz System for Assisted Learning* has significant potential to help students understand programming concepts and consequently achieve better marks in programming courses. I believe its implementation in Macquarie University undergraduate computing courses will benefit staff and students alike.

8. References

[1] K. Ala-Mutka, H-M Jarvinen, E. Lahtinen. *A study of the difficulties of novice Programmers*. ITiCSE, 14-18. 2005

[2] M. Bakoˇsov´a, L'. Cˇirka, M. Fikar, T. Hirmajer. *Automatic Generation of Quizzes in Matlab*. Slovak University of Technology in Bratislava Institute of Information Engineering, Automation, and Mathematics. ISBN: 978-80-227-2677-1. 2007.

[3] P. Brusilovsky, S. Sosnovsky. *Individualized Exercises for Self-Assessment of Programming Knowledge: An Evaluation of QuizPACK*. University of Pittsburgh. ACM Journal of Educational Resources in Computing, Vol. 5, No. 3. Article 6. September 2006

[4] T. Clear, A. Haataja, J. Meyer, J. Suhonen, S. Varden. *Dimensions of Distance Learning for Computer Education*. Proceedings of ITiCSE. ACM Press. 2000.

[5] S. Dekeyser, T. Yu Lee, M. de Raadt. *Computer Assisted Assessment of SQL Query Skills*. Department of Mathematics and Computing, University of Southern Queensland Australia. 18th Australasian Database Conference, Australian Computer Society. 2007.

[6] M. Evans, M. Joy, B. Muzykantskii, S. Rawles. *An Infrastructure for Web-Based Computer-Assisted Learning*. University of Warick. ACM Journal of Educational Resources, Vol. 2, No.4. Pages 1-19. December 2002.

[7] F. Griffin. *Designing randomised quiz questions for mathematics - a case study*. Department of Mathematics, Macquarie University, Sydney. Maths CAA Series, LTSN Maths, Stats & OR Network, University of Birmingham, UK, January 2004. Available online at http://ltsn.mathstore.ac.uk/articles/maths-caa-series/may2004/index.shtml - Accessed September 3, 2010.

[8] F. Griffin. *MacQTeX Randomised Quiz System for Mathematics*. Department of Mathematics, Macquarie University, Sydney. Maths CAA Series, LTSN Maths, Stats & OR Network, University of Birmingham, UK, January 2004. Available online at http://ltsn.mathstore.ac.uk/articles/maths-caa-series/jan2004/index.shtml - Accessed September 5, 2010.

[9] A. Gungor, A. Karahoca, D. Karahoca, I. Yengin. *Computer Assisted Active Learning System Development for Critical Thinking and Flow*. International Conference on Computer Systems and Technologies - CompSysTech'08. 2008.

[10] M. Heo. *A Learning and Assessment Tool for Web-Based Distributed Education*. Florida State University. CITC4'03. ACM. October 2003.

[11] D. Mason, D. Woit. *Enhancing Student Learning Through On-line Quizzes*. Ryerson Polytechnic University. SIGCSE, ACM. 2000.

[12] A. Radenski. *Digital Support for Abductive Learning in Introductory computing courses*. ACM, SIGCSE'07, March 2007.

[13] Moodle Team. *Moodle XML Format for Quiz Modules*. Available online at http://docs.moodle.org/en/Moodle_XML_format - Accessed 12 October 2010.

[14] *PHP Manual*. The PHP Group. Available online at http://www.php.net/manual/en/ - Accessed August 25 2010.

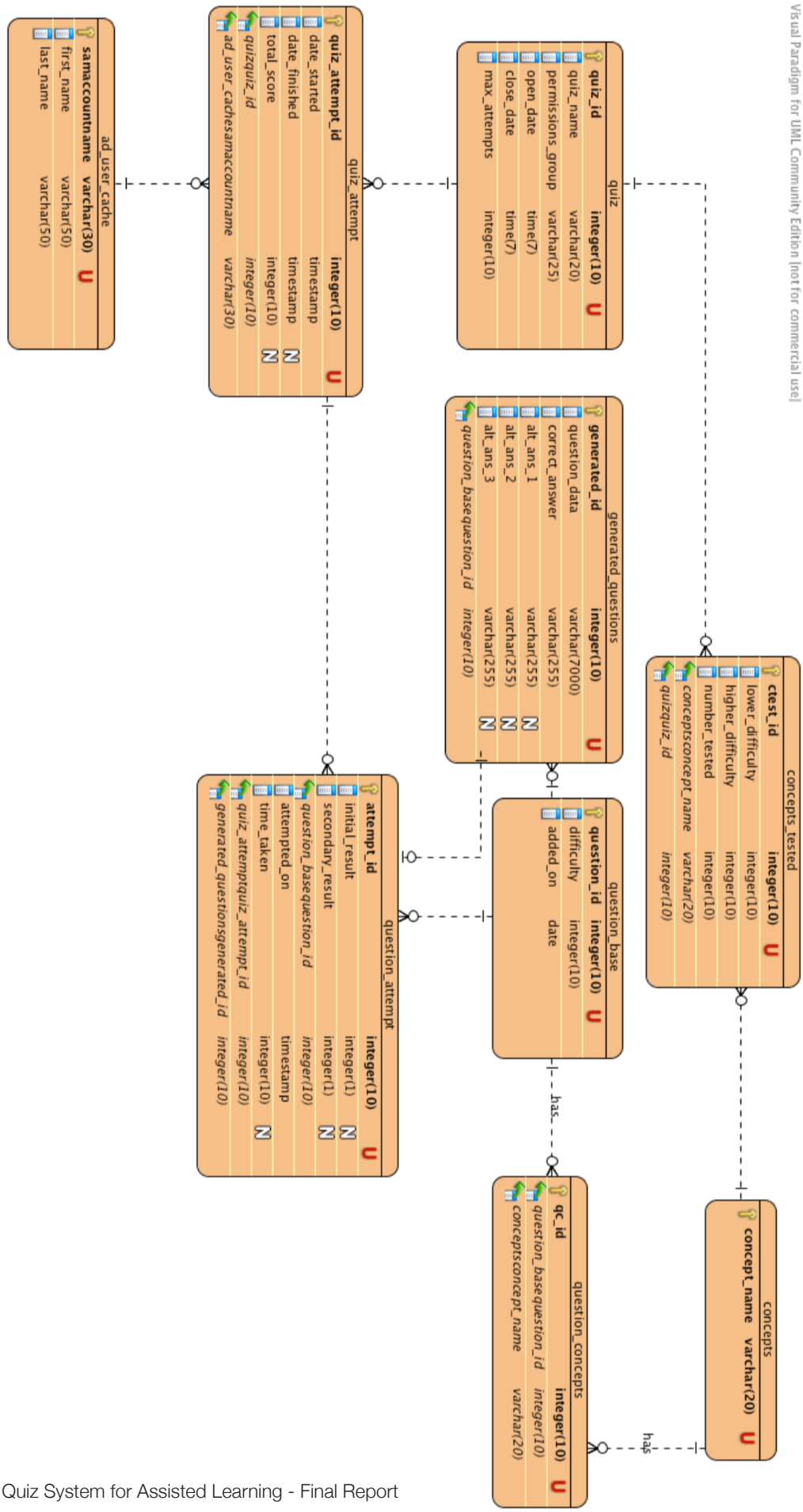
[15] *The Apache Manual* - Maximum concurrent connections. The Apache Software Foundation. Available online at http://httpd.apache.org/docs/2.2/mod/mpm_common.html#maxclients - Accessed October 25 2010.

[16] *Active Directory Users, Computers, and Groups*. Microsoft. Available online at http://technet.microsoft.com/en-us/library/bb727067.aspx - Accessed November 1 2010.

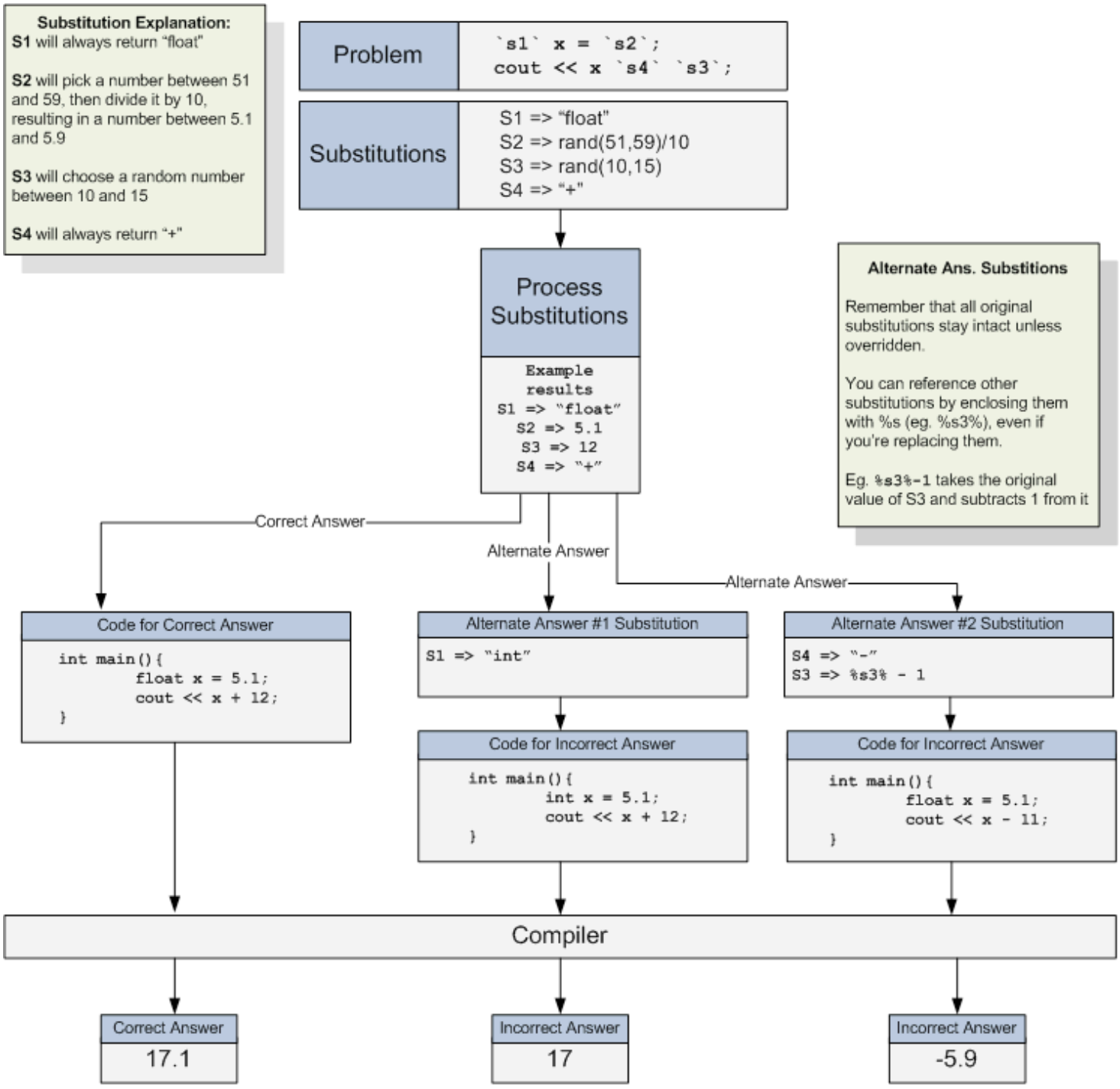
[17] S. Barnett, R Hyland. *LDAP Authentication with PHP for Active Directory.* Available online at http://adldap.sourceforge.net/ - Accessed November 1 2010.

[18] *JPGraph - PHP Driven Charts*. Asial Corporation. Available online at http://jpgraph.net/ - Accessed November 1 2010.

Appendix A - Entity Relationship Diagram



Appendix B - Alternate Answer Data Flow



Appendix C - Example XML Question Template

```
<question type="multiple">
        <estimated_time>220</estimated_time>
        <concepts>
                <concept>Queues</concept>
        </concepts>
        <difficulty>2</difficulty>
        <instructions>
                What is the given output of this program?
        </instructions>
        <problem>
#include &lt;iostream&gt;
#include &lt;string&gt;
#include &lt;queue&gt;
using namespace std;
int main(){
  queue&lt;string&gt; queueObject;
  queueObject.push("`s1`");
  queueObject.push("`s2`");
  queueObject.push("`s3`");
  cout &lt;&lt; "Contents of queue: ";
  for(int i = 0; i &lt; queueObject.size(); i++) {
    `s4`
    queueObject.push( queueObject.front() );
    queueObject.pop();}
  cout &lt;&lt; endl;
  `s7`
  while( !queueObject.empty() ) {
    `s5`
    `s6`}
  return 0;}
        </problem>
<substitutions>
        <substitution val="s1">randset(array("Deranged","Puppy","Smells","Insensitive","Ornament"))</substitution>
        <substitution val="s2">randset(array("Vampire","Bill","Drains","Sookie","Clean"))</substitution>
        <substitution val="s3">randset(array("Harry","Potter","Trains","Silvery","Colleagues"))</substitution>
        <substitution val="s4">return "cout &lt;&lt; queueObject.front() &lt;&lt; \" \";";</substitution>
        <substitution val="s5">return"cout &lt;&lt;\"Popping \"&lt;&lt;queueObject.front()&lt;&lt;endl;";</
substitution>
        <substitution val="s6">return "queueObject.pop();";</substitution>
        <substitution val="s7">return "";</substitution>
</substitutions>
<answers>
        <answer>
                <substitute val="s4">//Remove S4 Substitution;looks like the for loop never gets executed
                return "";</substitute>
                <description>It is important that you notice the FOR loop and the line "cout &lt; &lt;
queueObject.front()". This line gets the item at the front of the queue, and outputs it. The answer you chose was
missing that output. The next two lines in the for loop make sure that those elements are added back into the
queue.</description>
        </answer>
        <answer>
                        <substitute val="s5">
//Effectively remove the S5 substitution-looks like after you've iterated the array, there's nothing left to pop
                                return "";</substitute>
                        <description>It is important to understand that the FOR loop in this program not only
reads and pops items from the queue, but it also puts them BACK, so that when the WHILE loop happens, it appears
that a duplicate set of items is popped.</description>
        </answer>
        <answer>
                <substitute val="s7">
        //Making sure we actually understand what the pop function does. What if I pop a value out early?
                        return "queueObject.pop();";</substitute>
                <description>Revise Queue.pop() and Queue.push() functions.</description>
        </answer>
        </answers>
</question>
```

This is the problem definition.
This particular problem examines queues and is difficulty 2

This is the C++ Code.
All substitutions are enclosed by ` characters

This part tells the interpreter what to substitute to make this quiz randomised