StudTest – a platform supporting complex and interactive knowledge assessment

Vlado Glavinić¹, Marko Čupić¹, Stjepan Gros³

Faculty of Electrical Engineering and Computing, University of Zagreb¹

Key words: knowledge assessment, e-learning, authoring tool

Abstract:

This paper describes the model and prototype implementation of a knowledge assessment framework based on problem management components. In order to support student testing with complex problem types and enable usage of rich graphical user interfaces for solution inputting, we have developed an e-examination model in which a component that can generate complex questions and evaluate students' solutions with additional explanation generation, named prlet, is the core concept. The respective system implementation is described, which can operate under heavy loads.

1 Introduction

Computer based systems for supporting and enhancing faculty courses are used more and more each day (some of the more significant examples being [1], [2], [3]). However, since main goal of those systems is to be comprehensive, in a way that offers capabilities for course organization, course material repositories and student management and knowledge assessment, some of those capabilities are not expressive enough which introduces limiting factors for their usage. One notable representative is the assessment of students’ knowledge, which is typically based on quizzes with limited capabilities. Multiple choice questions are often used at university level to assess students' knowledge [13], [14], [15]. In this paper we will describe an open framework based on portable technologies and designed with extensibility in mind, specialized only for assessment of students knowledge. This framework will heavily relay on the concept of prlets (pronounced as “pearl-ets”) – a term coined in a similar way as the servlets, today widely accepted and used as a core Java based technology [4] for Web applications, and standardized by Sun [5]. It is also worth mentioning that a similar attempt was made at Ramapo College of New Jersey to design more-than-usually-capable problems known as problettes [6] which can be used in most Java enabled Web browsers in form of Java Applets.

This paper is organized as follows. In Section 2 we will give an overview of typical quizzing capabilities offered today. Section 3 will provide an overview of practical considerations on quizzes and tests in general. In Section 4 we will define a model of StudTest, and briefly describe its prominent components. In Section 5 we will present our prototype implementation of StudTest framework and discuss it briefly. Section 6 presents conclusion and future work directions.
2 Quizzes overview

Since the problems (of which quizzes are composed) are the core for knowledge assessment, in this section we will describe typical problem capabilities offered today. We will analyze four factors describing each problem: auto evaluation capability, dynamics capability, presentation randomization capability and multitechnology presentation capability. 

Auto evaluation capability means that problem can automatically evaluate student answer and determine its correctness. Representative of problem lacking this capability is essay-like problem, where student writes answer in standard natural language form. Representative of problem having this capability is ABC type question, where the correct answer is known in advance by the system. Some problems where student must enter textual answer can even have this capability (but in a limited sense), if the answer is limited to only few predetermined words, that can then be verified by means of a regular expression (although this approach has its issues with synonyms, typos etc.), or where multiple words can be offered requiring student to select the correct one (and checking dominantly his recall capabilities, and less his knowledge).

Dynamics capability discriminates problems that can utilize some form of template for question generation versus problems having statically preloaded question text (and depending on problem-type, possible answers). For example, typical ABC questions supported in many popular e-learning systems does not have this capability (question text and possible answers must be preloaded by a human), although they have presentation randomization capability, which means that the same question will (usually) not be presented in a same way to two students (for example, offered options will have randomized order of appearance). The simplest form of a problem having dynamics capability is the problem that can be stated in a pseudo language such as “What is the result of addition of \{a\} and \{b\}”, with correct solution given as “\{a\}+\{b\}” and constraints such as “\{a,b\} are integer in [0,50]”. More advanced dynamics capability is associated with problems capable to dynamically generate multimedia objects and incorporate them as part of questions (e.g. per student images).

Multitechnology presentation capability means that problem can be presented to user by a variety of technologies (e.g. through some local windows based application, through Web browser, through cell-phone etc.). It is widely lacking capability. This factor is also important for it determines how user can answer the question (by clicking/selecting something, by entering some text/number or perhaps by drawing something).

There are also some important factors concerning quizzes themselves: what problems will the quiz be composed of, how many questions will be asked, will (in)correctness of previous question have any influence on the selection of the next question? All of those questions lead to one common factor – adaptability. And if the quiz is adaptable, on what ground is this adaptability based? It can be based on some simple algorithm or on more complex algorithms and methods from the field of artificial intelligence.

During the phase of planning of our framework we chose not to fix any of those factors and capabilities, and to build a framework capable of supporting them all. Since the quizzing represents just one form of testing the students knowledge, in the remaining part of this paper we will use more general term – tests.

3 Practical considerations

Besides the problems themselves, there is a number of other considerations to be analyzed, resulting from real system usage (of which we have some experience, working with simultaneous groups of over 120 students, and total course population of over 1100 students). Security is an important issue – who will be able to access the tests and when. If testing is used in a supervised way for whole generation of students but in smaller groups at a time (e.g.
limited number of available computers) care must be taken to disallow access to tests to students that are not under staff supervision. Today, this is typically accomplished by password protection (communicated to students present in the examination room). To disable password leakage to outdoor students (e.g. through cell-phones), IP based control can also be utilized, and passwords can be changed.

Course policy is another issue that must be taken into account. Typical example is policy such as “Student can not get test X, if she has not passed test Y”, or “Student must pass test X, where the number of attempts is unlimited.” Also, some courses have following policy: “Student can solve test X as many times as she wants; we will grade her by her last attempt” (which is commonly used in attempt to ensure that students learned the course material).

Another example of course policy is “Test X can be taken for no longer than 15 minutes.”

Last issue we will analyze is scalability and heavy load handling. System must be correctly dimensioned, so that it can handle large total number of its users (e.g. thousands). But depending on specific course organization, situations are possible where many of students will use system simultaneously during time-constrained testing scheduled at fixed time (which can generate heavy peek-loads) – in those conditions it is critical that system offers small response times. This can be achieved in two ways – by building a clustered system with load-balancing support (more expensive) and/or by building a system based on asynchronous operations that can postpone less important operations during heavy load.

In order to offer both solutions, we have defined a framework that can easily be clustered, and that is based on asynchronous operations.

4 Model

To offer a variety of possibilities and capabilities (see Section 2), we have defined the concept of prlet, and constructed the rest of the framework to be a prlet container – a component based environment that executes prlets and supports pluggable objects. The general idea behind prlets is that they represent pluggable components which have public names and can be referenced globally (which means they can also be easily shared). They also contain the complete logic needed for problem editing, instantiation, and possibly evaluation. This framework operates with several categories of objects, as follows.

4.1 Core elements

The TestDescriptor represents description of a test – it enables selection and inclusion of container supported mechanisms (those are discovered dynamically by active plug-ins examination). TestDescriptor will enable inclusion and configuration of available security constraints, TestController (that determines, for example, whether the test is adaptive, which problems are to be included), and TestGrader (which determines score assignment policy).

The Test is a wrapper which is constructed for each user. Test can be visualized as a folder, containing all user’s attempts to solve specified TestDescriptor.

The TestInstance is one concrete test that was presented to the user. For each user and each TestDescriptor, the TestInstances are grouped into collections by means of Test.

ProblemType represents a way on which problem is (mentally) presented to the user. Typical problem types are single-correct-ABC-question, multiple-correct-ABC-question, input-textOrNumber-question, input-listOf-textOrNumber-question and CustomProblemPanel. The latter was defined in order to support problems that will be only presented in graphical user interface, and that must offer rich tools for inputting solution (for example, by drawing). In StudTest framework we have decided to separate this information from the prlet itself, in order to detach type presentation issues and problem logic issues, leaving only the latter as a
part of prlet. This separation also enables implementation of multiplatform presentation capability. Namely, when user’s client contacts StudTest, as a part of the handshaking process it must send Technology identifier telling the StudTest what technology for test presentation the client supports. Based on this parameter, StudTest can than select for each ProblemType an appropriate ProblemRenderer component, which knows how to present that kind of problem type to the user using her technology (which is notably most often HTML, where CustomProblemPanel type can easily be support by the means of Java applets).

Hence, the ProblemRenderer is a component used for the presentation of a problem of a specified type using selected technology. Thanks to this separation, in order to add support for new technologies, all that is needed is to implement additional set of renderers for that technology – no prlet will be aware of the change.

The Prlet is a component composed of following objects: ProblemGenerator, one or more ProblemEditors, ProblemInstantiator and ProblemEvaluator.

The ProblemGenerator stores basic information on prlet – its public name, its problem type and whether or not it can automatically evaluate answers.

The ProblemEditor is a component that allows customization of a problem template based on which concrete questions (later ProblemInstances) are created. ProblemEditor includes supported technology identifier, which is a shortcut we willingly took. Namely, we believe that the main concern of knowledge assessment systems should be a wide range of supported technologies for problem presentation. Editing of problem template parameters, however, can be supported in a smaller range of technologies (of which standard HTML should be mandatory), and for each technology a new editor must be written.

![Figure 1. Core objects of StudTest.](image)

The ProblemInstantiator is a component in charge of concrete problem generation, based on current parameters of problem template. Most of the power of presented framework lies exactly here – for problem instantiation we have separate component which can, in order to create new instance of a problem use anything it needs (e.g. communicate with other servers on the Internet, use Web services for help, etc.). Since ProblemInstantiator knows the type of problem it creates, all necessary data imposed by that type contract will be stored in ProblemInstances repository.
The *ProblemEvaluator* is a component that evaluates and generates comments on user’s solution, determines its correctness using a predefined measure, and generates correct solution (if this is calculable / supported, and no correct answer was provided by the user). Due to this responsibility division, evaluators will be able not only to implement complex algorithms themselves, but also to use other resources for evaluation purposes (such as contacting some other servers on the LAN or even the Internet, to use clusters prepared for necessary calculations, etc.).

For each problem type there is a mandatory interface that must be supported by all ProblemType implementations, consisting of methods for obtaining user’s help, obtaining explanation generated during evaluation process and obtaining information on correct solution: is it available (i.e. has evaluator generated one) and is it unique (if more correct solutions exists, we define mechanism for presentation of only one; namely, depending on the problem, situations are possible where the number of correct solutions is infinite). Relationships among those components are shown on Figures 1 and 2.

Many of mentioned components use private repositories for information storage. StudTest model defines Repositories as a name-distinguished collection of Repository objects. Each Repository object contains two separate containers: KeyRepository and AttachmentRepository. KeyRepository is a collection of key-value pairs, where keys are textual objects, and values arrays of bytes (so that any kind of content can be stored). AttachmentRepository is a collection of name-distinguished attachments (each having name, mime-type and content in form of associated byte array).

### 4.2 Other elements

All model elements described so far were core components of the framework directly associated either with tests or with problems. What remains is to define components for enforcing test security constraints and course policy, management of examination process and test instance grading.

Components for enforcing test security constraints and course policy are *TestStartCheckers*. These components can (at TestDescriptor creation time) be associated with TestDescriptor (by the person creating the test) and configured accordingly. The most important method those components offer is *isStartAllowed*, which checks whether user satisfies by it imposed
constraints. If she does, permission for start will be granted. If test designer decided to use more that one TestStartChecker, then start will be granted if and only if all checkers granted it. Foreseeable applications for checkers are: starting from allowed IP addresses, starting only if required password is known, starting at specified time frame (from some date to some date), starting during the predefined interval after supervisor explicitly enabled start, starting only if other test prerequisites are fulfilled (passed other tests) etc.

During the examination process, other components are used for examination process supervision – TestSupervisors. They can be associated with TestDescriptor (by a person creating the test) and configured accordingly. Those components have two tasks: to supervise process of test writing, and to generate status information for the user. Foreseeable application for supervisor is restriction of time during which the test must be solved, informing the user of the time remaining, etc.

Component in charge for management of the examination process (i.e. examination workflow) is the TestController. There can be many TestControllers plugged in a system, but during the TestDescriptor creation, person who defines it must select one of the available components, and configure it accordingly. TestController determines what is presented to the user, and when. Those decisions can be made dynamically. Namely, first time the user accesses its test, selected TestController is asked to determine what question(s) will be presented to the user at that point. When the user solves those questions, TestController is asked again what to do next. This way, depending on implementation of TestController various scenarios can be achieved, ranging from simple quizzes that determine all of its questions on first call, to adaptive intelligent tests that ask only a few questions, analyze evaluation results, ask additional questions with respect to those results, etc.

When test writing is done, it remains to calculate total score for that test. Recognizing the fact that this is very sensitive area, we have decided to model this process with additional component named Grader. During the evaluation of users solution correctness, ProblemEvaluator will assign correctnessMeasure parameter as a value from interval [0, 1], 0 being totally wrong and 1 being absolutely correct. Also, during the process of test writing, we expect that the user will be offered to enter her confidencyMeasure (a number from interval [0, 1]), or that default value of 1 will be used (however, the former should be preferred). Based on those parameters (correctnessMeasure and confidencyMeasure), and the information whether the user has solved the question or left it unsolved, various grading strategies can be adopted (positive/negative score, scoring proportional to correctness and confidence, etc.).

4.3 Helper elements

By analyzing many faculty courses and problems suitable for their students knowledge assessment, we have discovered an interesting fact that within given course many questions can be stated having in mind very few course-related concepts whose representation is typically rather complex. To illustrate this, let’s imagine a course on Computer Networks. A number of questions can be stated beginning with: “A computer network is shown on Figure 1. How ...”. However, generating an image which shows computer network, and has symbols for routers, hubs, switches, servers and regular computers, determining where and how to place each component is not a trivial task at all, and will be main stopper in creation of new problems (especially dynamic-ones, where each student can be given her own network). In order to foster the usage of computer generated problems we have provided a component-based facility that can alleviate these problems. The idea is to introduce a set of components called Helpers which can, based on given parameters, produce the required multimedia content (most often images). In the case of StudTest, those components have direct access to allowed ProblemInstance repository content, and can generate requested content based on the
parameters found in the repository. On the other hand, ProblemInstantiator knows what helper it will use, and where it must leave the necessary data (from helper contract), and during the instantiation process, instantiator has the facility to include call to the helper (which will than be executed during the problem presentation phase). In a context of HTML technology, this call to the helper will be rendered as an IMG tag, which will cause the browser to make an additional request to the server, which will in turn start the helper, which will generate requested content and finally deliver it back to the client.

5 Prototype StudTest implementation

StudTest framework briefly modeled in Section 4 was implemented in the Java programming language. Java technology was chosen for two reasons: first, up to day, only Java offers stable and portable platform for application development. Because of its broad acceptance and existence of Virtual Machine implementations for almost all widely used operating systems, Java was the only optimal choice. Second reason is the fact that we wanted to support graphically rich problems with complex user interface and complex solution inputting methods. Considering the tendency to move assessment systems to Web and HTML, again, only portable platform for those purposes is offered by Java Applets [7]. Additional reason is the fact that in order to support dynamical problems, some kind of scripting language is needed. Today, JavaScript is usually used as such language. JavaScript was initially introduced as simple scripting language (for client side simple evaluations and event handling) and was never meant to be fully flagged object oriented programming language. Instead, we have decided to base all of our “scripting” needs on regular and widely accepted object-oriented programming language [8] with modern language constructs and built-in support for concurrency.

Our implementation of StudTest model is illustrated on Figure 3. As can be seen, implementation is a standalone module that relies on database for data persistence (although it does not have to be relational database, thanks to persistence virtualization layer, see Figure 4), and offers its services to clients through developed connectors. Currently, only one connector is implemented – the TCP/IP based binary connector with connection pooling. This was done to improve the performance. However, other connectors are being developed as well, one of which is the Web Services connector that will expose StudTest functionality through Web Services over HTTP.

![Figure 3. Our implementation of StudTest model.](image-url)
Since our default persistence storage is MySql [9], which is a relational database management system, we needed an appropriate object-relational mapper. Instead of implementing this from scratch, we decided to use Hibernate 2 [10] which is a well spread and recognized O/R mapper for Java.

To allow easier system distribution and to enable better peek-load handling, most of operations in StudTest are implemented as asynchronous, and communication with all components is hidden behind suitable interfaces, that will allow easy replacement / reimplementation of components (when necessary). Most notably, there are two distinguished queues: the ProblemInstantiation queue and the ProblemEvaluation queue, both being implemented as priority queues. When there is a need to instantiate problems, the process is not started immediately – instead, instantiation request is added to the ProblemInstantiation queue. Similarly, when there is a need to evaluate a problem instance, request is added to the ProblemEvaluation queue. During system setup, configured number of InstantiatorWorkers and EvaluatorWorkers is started. Those workers continually read requests from appropriate queues, and execute them. When, from some reason, a large number of requests are generated, instead of a system collapse, those requests will be processed little by little, and the system will continue to function normally.

This design can also support scenarios of primitive and dedicated clustering configurations, where both queues can be exposed over TCP/IP connectors, and in which additional StudTest systems can be setup and dedicated to execution of instantiation and evaluation requests, while main StudTest server could handle examination workflow operations and users answers storage. Even more, since communication with Helpers also goes through well defined interfaces, it is possible to setup additional systems for Helper execution (since they can consume large amount of memory, during generation of multimedia contents).

### 5.1 Implemented components

From components that extend system capabilities, so far we have implemented a number of TestStartCheckers, one TestSupervisor, one TestGrader and one TestController. We implemented them as circumstances required, and currently developed components cover all of our needs for two large courses on the Faculty of Electrical Engineering and Computing.
(Digital Electronics and Digital Logics), as well as for few smaller courses (Intelligent Systems, Artificial Intelligence and Interactive Computer Graphics).

The TestStartCheckers we have implemented are the following: QueueStartChecker (requires student to register for test and provides simultaneous enabling of registered test instances), TimeFrameStartChecker (allows the test to start within the predefined time period), TimeWindowStartChecker (works with QueueStartChecker and disables start of test writing after predetermined amount of time passes since simultaneous enablement occurred), IPAddressStartChecker (allows the test to start only from selected ranges of IP addresses), PasswordProtectionStartChecker (enables start of test only if student knows the correct password) and PassedTestPrerequisiteStartChecker (enables start of test only if prerequisite tests were solved and passed).

We have implemented only one TestSupervisors – TestDurationSupervisor (configurable to allow fixed time for solving beginning at the moment the student started the test, or fixed absolute end moment as a date and time when test must be finished; in both configurations it announces information on remaining time up to the end of the test).

We have implemented one TestController, offering the following capabilities:

- static problem selection from given problem group (which can contain subgroups, or even exclusive subgroups),
- fixed number of presented questions,
- configuration for number of questions to display at once,
- forward/backward navigation (with possibility to turn off backward navigation),
- switch on/off for direct navigation to each question,
- maximum achievable score for test,
- threshold for test passing, and
- switch on/off for enabling of multiple solution attempts.

The last was the functionality we didn't anticipate as necessary from the beginning, but it emerged when Bologna process was introduced at our Faculty, and brought the concept of homework back to the examination procedures. Homework is a specific challenge, since it breaks apart typical examination procedure where student opens her test, solves it and closes it (all under heavy supervision of all available and configured security mechanisms). Namely, the idea of homework is to enable students to access their homework (which is, from the standpoint of the system, just another test), solve some questions, suspend solving process, resume it the next day, etc., within predetermined period of time (e.g. for one week).

However, due to a proper model and the system design, this functionality was easily added.

We have completed one implementation of Graders, offering rich configuration capabilities through the use of a simple scripting language, exemplified in Figure 5.

```plaintext
if $isSolved then
    if $isCorrect then
        return 10;
    else
        return -2;
    end if;
else
    return 0;
end if;
```

Figure 5. Simple grader configuration.
5.2 Integration with other systems

StudTest is a standalone module for user knowledge assessment – and nothing more. StudTest does not define nor provide any technology or connector through which user could directly work with StudTest (e.g. from Web browser). Instead, in order to be used, StudTest must be included into some other system that offers user interface, and communicates with StudTest through its connector. Typical and targeted system for StudTest inclusion is a Web based e-learning system, or a simpler Web based course management system. It is their task to provide suitable user interface, and through that user interface students will solve given tests. Namely, even when client accesses the StudTest and claims to use HTML technology, appropriate TestRenderer (a component that renders whole test in a given technology) will not create the whole Web page. Instead, with respect to general form of HTML document shown in Figure 6, it will deliver through its connector two chunks of HTML page (one to be included in HEAD part of document, and other to be included in BODY part of document). StudTest client is than free to add all surrounding data or adjust the look&feel of generated HTML document, before it sends it back to the user.

In order for clients to use and communicate with StudTest, they must utilize client side of the connector, which was also written in Java. And to further simplify StudTest utilization, we have prepared simple ready-to-be-used Servlet for inclusion.

We have tested StudTest and connectors with our Course Management System known as Nescume [11]. Through it, five generation of students passed (several thousands students) without any serious objections nor complains.

5.3 Support for prlet development

Since natural environment for prlets is prlet container, which is a large and complex environment, development of prlets typically poses some challenges. Namely, before deploying prlet into the prlet container, prlet should be finished, tested and ready. To support easier prlet development, we have also prepared prlet development framework, that helps in the process of prlet development, by simulating prlet lifecycle: creation of a new problem template, editing of an existing problem template, a problem instantiation, presentation of a problem to the user and problem solving (by user – in this case, the developer) and finally a problem instance evaluation and generation of comments and correct solution (if available/supported by prlet).

5.4 Example of developed prlets

An example of developed prlets is shown on Figure 7. For each student a random Boolean function is generated and displayed (visible in presented question). The student must construct a CMOS circuit which implements given function. For purposes of circuit construction, problem is shown through Java Applet with simple program for schema drawing. Elements

```html
<HTML>
<HEAD>
 <!-- head part of document -->
</HEAD>
<BODY>
 <!-- body part of document -->
</BODY>
</HTML>
```

Figure 6. Structure of HTML document.
(MOSFETs, inputs, outputs etc. are accessible from popup menu). During the evaluation phase, correctness of solution is checked using CMOS simulator. In this case, a code that creates question, allows student to solve it and evaluates its correctness comprises a single prlet.

Another example (Figure 8) shows random PLA-based circuit rendered by an appropriate Helper, where student is asked to analyze in which cycle the realized sequential circuit
counts. During problem construction, for each student prlet randomly creates one cycle, programs PLA based circuit and prepares data needed for associated helper to draw its schematics. Then, it randomly chooses additional three cycles, and present itself to a student as a Single Choice question.

6 Conclusion and future work

The StudTest was developed in an effort to offer rich and complex examination capabilities of generated problems and to be specialized subsystem for knowledge assessment, independent of user technology. It was designed with the following goals in mind: extensibility, good peek-load handling and scalability. More important, it presents a well worked out model based on which prototype implementation was built and successfully tested. This platform has been successfully used for the last five years, and by the several thousand students. It has been used on five university courses: Digital Electronics, Digital Logics, Intelligent Systems, Artificial Intelligence and Interactive Computer Graphics, each posing some specific requirements. For example, on Digital Electronics and Digital Logics it was used for tests at the beginning and the end of the laboratory exercises, and also for students’ homework.

Because of its advanced graphical capabilities, we were able to implements prlets such as one requiring students to draw a CMOS schema of a circuit implementing a randomly generated Boolean function, and verify the correctness of the design automatically. Since students accessed tests through Nescume, which is Web based, those complex problems were presented to students by the means of Java Applets. On those courses, we have developed over 90 prlets used in homeworks, and about 300 static problems used in laboratory exercises.

As a direction to future work, we plan to work out and implement better clustering support and implement an adaptive TestController that would use an ontologically described course structure and its relationships to the existing problems (based on RDF and RDFS) for problem selection. Support for reasoning about this knowledge will probably be obtained through the Sesame system [12].

7 Acknowledgments

This work has been carried out within project 036-0362980-1921 Computing Environments for Ubiquitous Distributed Systems funded by the Ministry of Science, Education and Sport of the Republic of Croatia

References:

[5] JSR-000053 Java Servlet 2.3 Specification

Author(s):

Vlado Glavinić, prof.dr.sc.
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
Vlado.Glavinic@fer.hr

Marko Ćupić, mr.sc.
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
Marko.Cupic@fer.hr

Stjepan Groš, mr.sc.
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
Stjepan.Gros@fer.hr