

# Assessment Delivery Engine for QTIv2 Tests

Gary Wills, Jonathon Hare, Jiri Kajaba, David Argles, Lester Gilbert and David Millard.

Learning Societies Lab, University of Southampton, UK.

## Abstract

The IMS Question and Test Interoperability (QTI) standard has not had a great take-up in part due to the lack of tools. In the 2006 JISC Capital, three Assessment projects were commissioned: item authoring, item banking, and QTI-compliant test delivery. This paper describes the 'ASDEL' test delivery engine, focusing upon its architecture, its relation to the item authoring and item banking services, and the integration of the R2Q2 Web service. The project first developed a java library to implement the system. This will allow other developers and researchers to build their own system or take aspects of QTI they want to implement.

## 1. Introduction

Formative assessment aims to provide appropriate feedback to learners, helping them gauge more accurately their understanding of the material set. It is also used as a learning activity in its own right to form understanding or knowledge. It is something lecturers/teachers would love to do more of but do not have the time to develop, set, and then mark as often as they would like. A formative e-assessment system allows lecturers/teachers to develop and set the work once, allows the learner to take the formative test at a time and place of their convenience, possibly as often as they like, obtain meaningful feedback, and see how well they are progressing in their understanding of the material. McAlpine [9] also suggests that formative assessment can be used by learners to *"highlight areas of further study and hence improve future performance"*. Steve Draper [10] distinguishes different types of feedback, highlighting the issue that although a system may provide feedback, its level and quality is still down to the author.

E-learning assessment covers a broad range of activities involving the use of machines to support assessment, either directly (such as web-based assessment tools, or tutor systems) or indirectly by supporting the processes of assessment (such as quality assurance processes for examinations). It is an important and popular area within the e-learning community [4, 1, 2]. From this broad view of e-learning assessment, the domain appears established but not mature, as traditionally there has been little agreement on standards or interoperability at the software level. Despite significant efforts by the community, many of the most popular software systems are monolithic and tightly coupled, and standards are still evolving. To address this there has been a trend towards Service-Oriented Architectures (SOA). SOAs are an attempt to modularise large complex systems in such a way that they

are composed of independent software components that offer services to one another through well-defined interfaces. This supports the notion that any of the components could be 'swapped' for a better version when it becomes available. A SOA framework are being used as a strategy for developing frameworks for e-learning [3, 5],

A leading specification has emerged in Question and Test Interoperability (QTI) developed by the IMS Consortium. The QTI specification describes a data model for representing questions and tests and the reporting of results, thereby allowing the exchange of data (item, test, and results) between tools (such as authoring tools, item banks, test constructional tools, learning environments, and assessment delivery systems) [8]. Wide take-up of QTI would facilitate not only the sharing of questions and tests across institutions, but would also enable investment in the development of common tools. QTI is now in its second version (QTIv2), designed for compatibility with other IMS specifications, but despite community enthusiasm there have been only a few real examples of QTIv2 being used, with no definitive reference implementation [6,7].

In this paper we firstly give an overview of the QTI specification and the R2Q2 project. Secondly, the architecture and the tools developed during the ASDEL project are described in Section 4. In Section 5 we describe the rational for first building a Java library for QTI. Finally, in Section 6 we presented a discussions and conclusion to this work.

## 2. QTI

The IMS QTI Specification is a standard for representing questions and tests with a binding to the eXtended Markup Language (XML, developed by the W3C) to allow interchange. An example of a simple multiple choice question illustrates the core elements: the *ItemBody* element declares the content of the question itself, the *ResponseDeclaration* declares variables to store the student's answer, and the *OutcomeVariables* element declares other variables, which will have their value calculated based on the students answer.

The QTI specification intentionally makes a big distinction between individual questions (QTI *items*) and the whole test (the QTI *assessmentTest*). JISC has funded two projects for rendering and processing QTI v2 xml. The first, R2Q2, developed rendering and responding engine for individual items. The second, ASDEL, developed tools for rendering complete assessments (using R2Q2 as the question renderer), in addition to providing support for other parts of the specification that are related to assessments, such as reporting.

In R2Q2 we focused on rendering and responding to 16 different types of interactions described in version 2 of the QTI specification (QTIv2). These are:

- |                   |                       |
|-------------------|-----------------------|
| 1) Choice         | 2) Hotspot            |
| 3) Order          | 4) Select point       |
| 5) Associate      | 6) Graphic            |
| 7) Match          | 8) Graphic Order      |
| 9) Inline Choice  | 10) Graphic Associate |
| 11) Text Entry    | 12) Graphic Gap Match |
| 13) Extended Text | 14) Position object   |

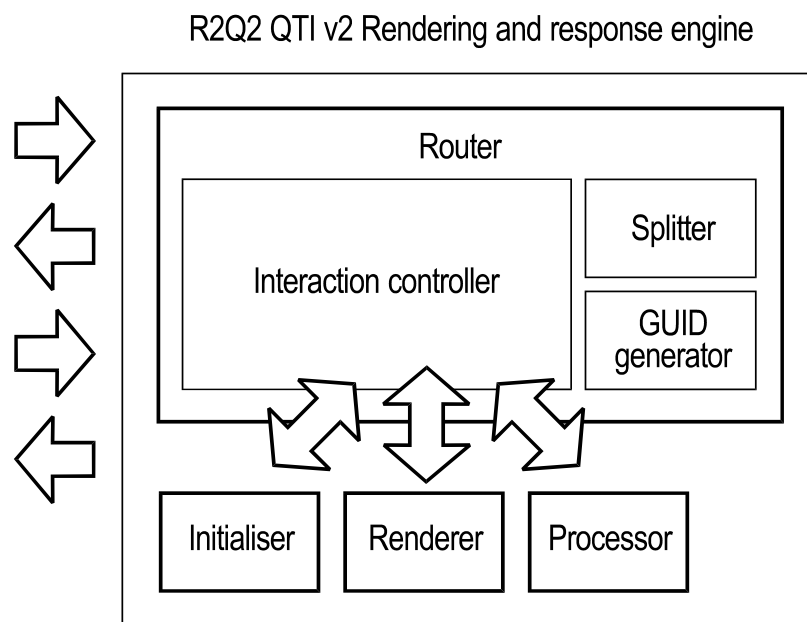
The different question types can be authored as templated or adaptive questions, providing an author with numerous alternative methods for writing questions appropriate to the needs of the students. Templated questions include variables in their item bodies that are instantiated when a question is rendered (for example, inserting different values into the text of maths problems). Adaptive questions have a branching structure, and the parts that a student sees depends on their answer to previous parts of the branch. In total these allow at least sixty-four different possible combinations of question types. In addition, the specification allows for any number of (possibly different) interactions within a single question.

### 3. R2Q2

As described in the previous section, the R2Q2 project developed a tool for processing individual QTI questions. The R2Q2 service allows a student to view a question, provide an answer, and then view any embedded feedback. The R2Q2 engine (see Figure 1) is a loosely coupled architecture comprising of three interoperable services. All the interactions with and within the R2Q2 engine are managed by an internal component called the Router.

The Router is responsible for parsing and passing the various components of the item (QTIv2) to the responsible web services. It also manages the interactions of external software with the system, and it is therefore the only component that handles state. This enables the other services to be much simpler, maintaining a loosely coupled interface but without the need to exchange large amounts of XML.

The Processor service processes the user responses and generates feedback. The Processor compares the user's answer with a set of rules and generates response variables based on those rules. The Renderer service then renders the item (and any feedback) to the user given these response variables.



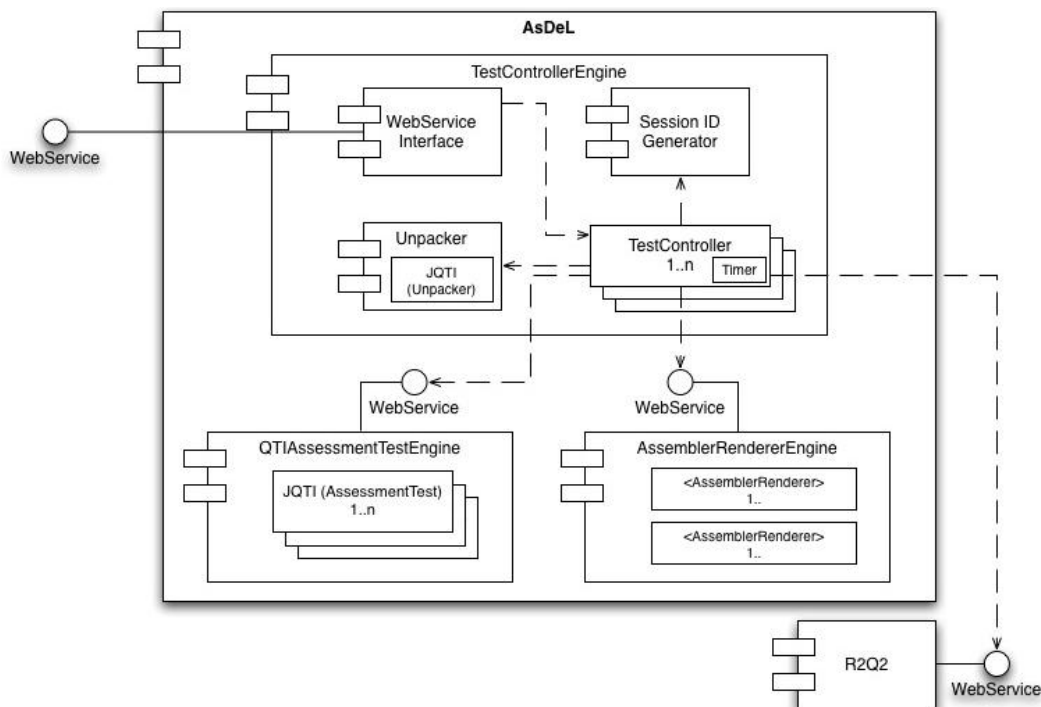
**Figure 1. The R2Q2 Architecture.**

#### 4. ASDEL

The ASDEL project aimed to build a tool for delivering QTI assessments. This involves a number of tasks, from assembling and rendering a sequence of questions (which may have logic to control the sequence), to collating results from each question and generating a report. The project was co-funded with the two other assessment projects in the JISC Capital Programme call:- item banking (Cambridge: Minibix) and item authoring (Kingston: AQuRate).

The QTI specification details how a test is to be presented to candidates, the order of the questions, the time allowed, etc. The ASDEL project built an assessment delivery engine to the IMS QTI 2.1 specification, called the ASDEL *playr*. The *playr* tool can be deployed as a stand-alone web application or as part of a Service Oriented Architecture enabled Virtual Learning Environment or portal framework.

The core components of the ASDEL system were built around a Java library called JQTI. The JQTI library enables valid QTI assessment XML documents to be interpreted and executed. The library also provides auxiliary services like the handling of QTI content packages and the provision of valid QTI reports. Figure 2 presents a conceptual overview of the *playr* and shows how the library integrates.



**Figure 2. Architecture for the ASDEL *playr* Assessment Delivery system.**

The AssemblerRenderingEngine part of the system is responsible for the assembly and rendering of output (i.e. questions and associated rubric). Initially, only an XHTML renderer has been developed; however, the design of *playr* will enable different renderers to be plugged in.

The ASDEL project integrated with the other projects in the JISC Capital Programme call on item banking (Cambridge: Minibix) and item authoring (Kingston: AQuR@te) to provide a demonstrator as shown in Figure 3. Together the three projects tell an

end-to-end story: AQuR@te will allow people to author items, which are stored in MiniBix. A test will incorporate these items and will be played through the ASDEL *playr*.

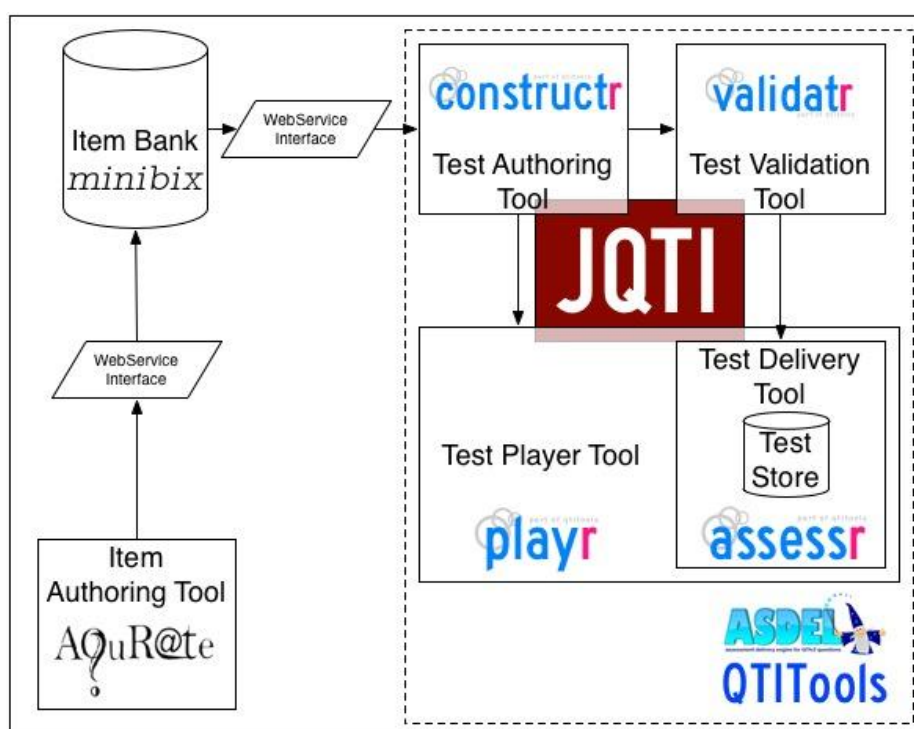
Most VLEs provide tools for assessment construction and delivery, and there is no intention to replace them. Instead, the projects seek to provide a lightweight suite of tools that early adopters may use to construct QTI-compliant tests and to manage delivery in a formative setting.

In addition to the *playr* tool, a number of complementary tools were developed by the ASDEL project. The *Validatr* tool provides validation of assessments and also gives indications of any error. Similar to an Integrated Design Environment for writing program code, the *Validatr* will also allow experienced users to correct the XML of the test. As can be seen from Figure 4, the *Validatr* has a visual front end that allows users to visualise the structure of the test and the different paths students can take through the tests.

The test player tool only delivers the test, so the *Assessr* tool manages the test for the lecturer or teacher. Lecturers can upload a class list from a spreadsheet, schedule the test, put embargos on the release of the test information, etc.

The *Assessr* (see Figure 5) sends a token and a URL for the test to each student. The students can then log into the *playr* using the token and take the test. The *Assessr* allows the academic to see which test they have set, who has taken them and QTI reports from everyone who has taken the test.

An extremely lightweight test construction tool has been developed, called a *Constructr*. This distinguished from item authoring since it simply allows a lecturer to select a pool of questions from an item bank and put them into a basic test.



**Figure 3. Integration of the ASDEL, AQuR@te Item Authoring (Kingston) and MiniBix, Item Banking (Cambridge).**

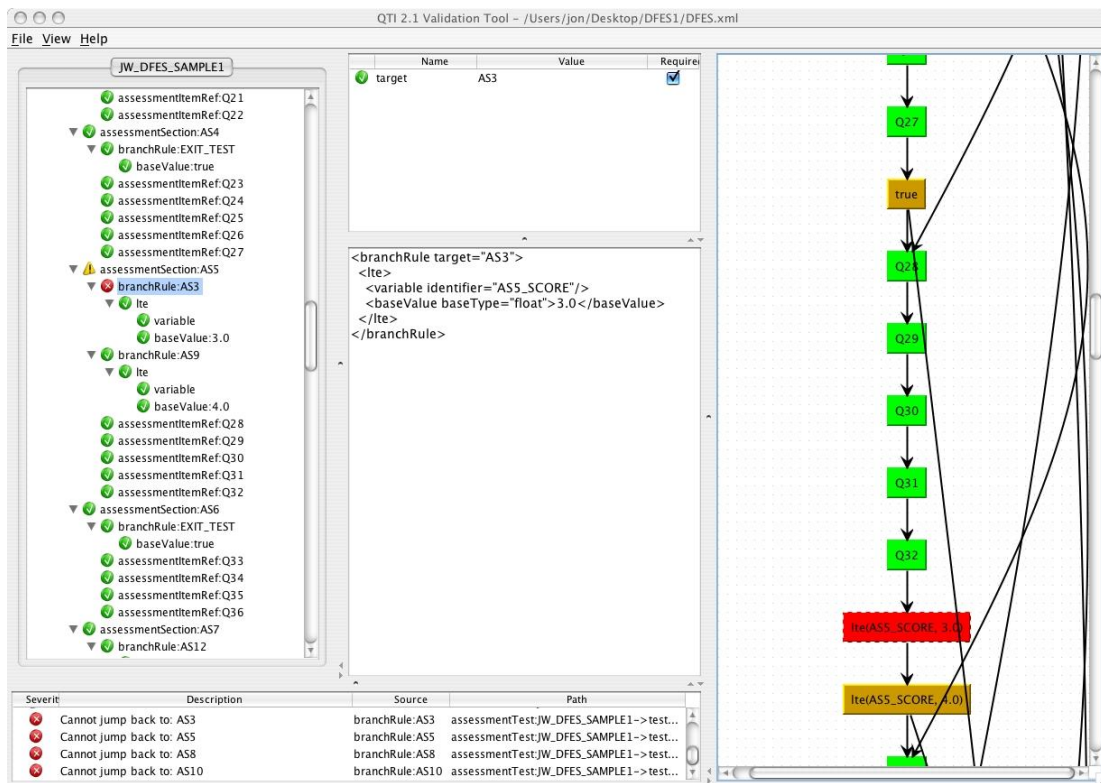


Figure 4. Validatr screenshot

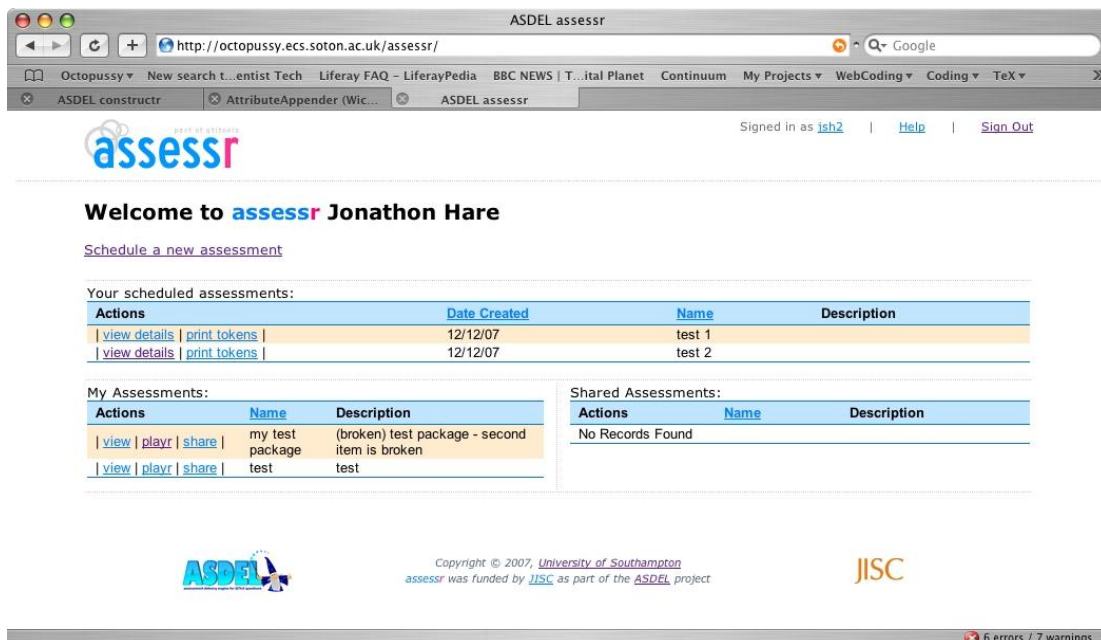


Figure 5. Assessor main screen.

## 5. JQTI: Why build a library first.

In this section we reflect on some of the issues and factors that needed to be considered in implementing a software library for the QTI specification.

The core of the ASDEL software is a library we are calling JQTI. JQTI is essentially an interpreter for IMS QTI v2.1 xml. QTI xml is rather unlike most xml documents as it doesn't only contain data, but also instructions. These instructions determine how tests and items are presented, processed and evaluated. Basically the QTI specification defines a programming language that just happens to be expressed in the form of an xml document. In so far as the ASDEL project goes, the scope is for JQTI to implement all of the parts relevant to the AssessmentTest class, although we hope in the future to add the remaining (AssessmentItem) classes and retrofit JQTI into R2Q2.

When faced with the implementation of JQTI we had two options to consider; we could either use a binding technology such as JAXB or Castor to bind the QTI xml schema to a set of automatically generated java classes, or we could write the whole library from scratch, using a DOM parser to parse the xml. XML binding technologies are great if you are binding to xml containing data, however they are slightly problematic when the xml contains instructions that need to be evaluated. Basically every automatically generated class would have to be manually modified to have a "behaviour" added to it so that it could be evaluated. Another problem with binding to the xml schema is that the schema is not nearly as expressive as the full QTI specification document — it is possible to have an xml document that validates against the schema, but is not valid QTI! It is for these reasons that we decided to go down the "custom classes + DOM parser" root in our implementation of JQTI.

A comprehensive library for handling QTI needs to perform two core operations; it needs to be able to generate QTI xml, and it needs to be able to parse/evaluate QTI xml. The library should not be responsible for the actual rendering of items/assessments, although it should provide relevant hooks to getting the required information needed for rendering out. The reason for this is that the specification itself is agnostic towards how content should be rendered (even though most implementations so far have rendered xhtml).

QTI xml is more of a programming language than a data-format. This fact means that there are some very special considerations that need to be taken into account when thinking about the design and implementation of a QTI library. Perhaps the biggest of these considerations is that it is possible (and frankly rather easy) to write a QTI xml document that is completely syntactically valid according to the QTI xml schema, but is not syntactically or semantically correct according to the specification. As an example of a syntax error, consider the following xml fragment:

```
<equal toleranceMode="relative">
  <baseValue baseType="float">1.0</baseValue>
  <baseValue baseType="float">1.0</baseValue>
</equal>
```

This xml will validate correctly against the QTI xml schema, however it is not valid against the specification because the element is missing the *tolerance* attribute that is required because the toleranceMode is relative. An example of a semantic error is referring to another element (for example in the target of a branchRule) using an identifier that does not actually exist.

This consideration mandates that the library must be able to validate the syntax of the QTI xml documents that it reads (in a more comprehensive manner than by simply validating the xml against the schema) and also assess the semantic correctness of the document. The semantic correctness is important, because the chance of any errors or exceptions being thrown during the execution of a test or item needs to be minimised - unfortunately it is impossible to check every possible error case because many things during the processing the the xml will rely on user input. If we want to check QTI xml for semantic correctness (at least as far as possible), then we essentially want to perform static analysis on the xml document in order to verify that it will work correctly as it is executed.

In terms of the implementation for processing and evaluating a QTI xml document, there are two possible ways of going about this; either parsing the data on a line-by-line basis and performing steps as required (for example by user response), or by reading in all the data and constructing an object tree. The first option has the advantage of lower memory consumption, however it has disadvantages as it would make the implementation of syntactic and semantic validation (c.f. static analysis) difficult and would also make moving around (i.e. backward/forward through a test) an absolute nightmare to implement correctly.

The class hierarchy in the library also needs to be considered. The specification provides some hints as to how QTI classes are related, but is by no means an implementation guide. As an example, many of the classes defined in the QTI specification are implemented in our JQTI library, however, the hierarchy is often a little different - i.e. all our java classes that are related to QTI classes inherit from a common abstract XmlObject class. Another consideration is that some classes defined in the specification are not relevant to the processing and evaluation of the xml document, and only serve as hints to the renderer (i.e. the xhtml classes used in the spec). These classes possibly don't need to have any concrete implementation associated with them. The QTI specification also serves as a good pointer as to how to breakdown the class structure into a suitable granularity - for example it doesn't really make sense to try and implement all of the expression classes in a single class (because there are so many of them, and because some are rather complex), but rather follow the specification and make all the expressions individual classes that inherit a common abstract expression class that contains methods that can be overridden for evaluation of the expressions.

The final thing that the library requires is a good testing framework. The QTI specification forms a good basis for determining a set of functional requirements for each class within the specification. The library implementation can make use of this for determining a set comprehensive unit tests for individual components, and also for determining when runtime exceptions should be thrown.

In summary, a good QTI library implementation needs to consist of a set of custom classes that implement the functionality of the QTI specification (i.e. items and assessments can be run, evaluated, validated, etc), and that also bind to the xml (so that tests and items can be read in and written out). The library also needs to be backed by a comprehensive test suite that validates that it conforms to the word of the specification, and handles runtime errors in a systematic way though the use of exceptions.



## 6. Conclusions

At a recent conference, the UK assessment community confirmed that kick-starting the use of the IMS Question and Test Interoperability version 2 specifications was a high priority. The conference concluded that there needed to be a robust set of tools and services that conformed to the QTIv2 specification to facilitate this migration.

R2Q2 is a definitive response and rendering engine for QTIv2 questions. While this only deals with an item in QTI terms, it is essential to all processing of QTI questions and so forms the core component of all future systems. Due to the design and use of internal Web services, the system could be enhanced if required. So while every effort has been made to ensure this service can be dropped into future systems, if necessary it can be changed to suit any application

In the ASDEL project we built an assessment delivery engine to the IMS Question and Test Interoperability version 2.1 specifications. Like R2Q2 this is a Web service based system that can be deployed as a stand-alone web application or as part of a Service Oriented Architecture enabled Virtual Learning Environment or portal framework. The engine itself cannot function alone so a small set of lightweight support tools have also been built. The engine provided in combination with the tools:

- Delivery of an assessment consisting of an assembly of QTI items, with the possibility that the assessment is adaptive and that the ordering of questions can depend on previous responses,
- Scheduling of assessments against users and groups,
- Rendering of tests and items using a web interface,
- Marking and feedback, and
- A web service API for retrieving assessment results.

In summary, we have provided a small set of lightweight tools that will enable a lecturer or teacher to manage a formative assessment using the World Wide Web quickly.

## 7. Acknowledgements

This work was funded in the UK by the Joint Information Systems Committee (JISC).

## 8. References

- [1] Bull, J., and McKenna, C. Blueprint for Computer Assisted Assessment. Routledge Falmer, 2004.
- [2] Conole, G. and Warburton, B. "A review of computer-assisted assessment". ALT-J Research in Learning Technology, vol. 13, pp. 17-31, 2005.
- [3] Olivier, B., Roberts, T., and Blinco, K. "The e-Framework for Education and Research: An Overview". DEST (Australia), JISC-CETIS (UK), [www.e-framework.org](http://www.e-framework.org), accessed July 2005.
- [4] Sclater, N. and Howie K. User requirements of the "ultimate" online assessment engine, Computers & Education, 40, 285–306 2003.
- [5] Wilson, S., Blinco, K., and Rehak, D. Service-Oriented Frameworks: Modelling the infrastructure for the next generation of e-Learning Systems. JISC, Bristol, UK 2004.

[6] APIS [Assessment Provision through Interoperable Segments] - University of Strathclyde-(eLearning Framework and Tools Strand) <http://www.jisc.ac.uk/index.cfm?name=apis>, accessed 30 April 2006.

[7] Assessment and Simple Sequencing Integration Services (ASSIS) – Final Report – 1.0. <http://www.hull.ac.uk/esig/downloads/Final-Report-Assis.pdf>, accessed 29 April 2006.

[8] IMS Global Learning Consortium, Inc. IMS Question and Test Interoperability Version 2.1 Public Draft Specification. <http://www.imsglobal.org/question/index.html>, accessed 9 January 2006.

[9] McAlpine, M. Principles of Assessment, Bluepaper Number 1, CAA Centre, University of Luton, February 2002.

[10] Draper, S. W. Feedback, A Technical Memo Department of Psychology, University Of Glasgow, 10 April 2005: <http://www.psy.gla.ac.uk/~steve/feedback.html>.