IMS Question and Test Interoperability Assessment Test, Section, and Item Information Model

Version 2.1 Public Draft Specification

Document Name: IMS Question and Test Interoperability Assessment Test, Section, and Item Information Model

---

Date Issued: 8 June 2006

Caution: this specification is incomplete in its current state. The IMS QTI project group is in the process of evolving this specification based on input from market participants. Suppliers of products and services are encouraged to participate by contacting Mark McKell at mmckell@imsglobal.org. This specification will be superseded by an updated release based on the input of the project group participants.

Please note that supplier's claims as to implementation of QTI v2.1 and conformance to it HAVE NOT BEEN VALIDATED by IMS GLC. While such suppliers are likely well-intentioned, IMS GLC member organizations have not yet put in place the testing process to validate these claims. IMS GLC currently grants a conformance mark to the Common Cartridge profile of QTI v1.2.1. The authoritative source of products and services that meet this conformance is contained in the IMS online product directory
http://www.imsglobal.org/ProductDirectory/directory.cfm

Thank you for your interest in and support of IMS QTI.

IPR and Distribution Notices

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the specification set forth in this document, and to provide supporting documentation.

IMS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on IMS's procedures with respect to rights in IMS specifications can be found at the IMS Intellectual Property Rights web page: http://www.imsglobal.org/ipr/imsipr_policyFinal.pdf.

If you wish to copy or distribute this document, you must complete a valid Registered User license registration with IMS and receive an email from IMS granting the license to distribute the specification. To register, follow the instructions on the IMS website: http://www.imsglobal.org/specificationdownload.cfm.

This document may be copied and furnished to others by Registered Users who have registered on the IMS website provided that the above copyright notice and this paragraph are included on all such copies. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to IMS, except as needed for the purpose of developing IMS specifications, under the auspices of a chartered IMS project group.

Table of Contents

1. Introduction

This document describes the information model for items, sections, and whole tests used in assessment.

2. References

CMI
IEEE 1484.11.1, Standard for Learning Technology - Data Model for Content Object Communication
ISO11404
ISO11404:1996 Information technology — Programming languages, their environments and system software interfaces — Language-independent datatypes
Published: 1996
ISO8601
ISO8601:2000 Data elements and interchange formats – Information interchange – Representation of dates and times
Published: 2000
ISO_9899
ISO/IEC 9899:1999 Programming Languages - C
MathML
Mathematical Markup Language (MathML), Version Version 2.0 (Second Edition)
http://www.w3.org/TR/2003/REC-MathML2-20031021/
Published: 2003-10-21
RFC2045
RFC 2045-2048 Multipurpose Internet Mail Extensions (MIME)
RFC3066
RFC 3066 Tags for the Identification of Languages
H. Alvestrand
http://www.ietf.org/rfc/rfc3066.txt
Published: 2001-01
URI
RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax
Published: 1998-08
XHTML
XHTML 1.1: The Extensible HyperText Markup Language
XHTML_MOD
XHTML Modularation
http://www.w3.org/MarkUp/modularization
XINCLUDE
XML Inclusions (XInclude) Version 1.0
http://www.w3.org/TR/xinclude/
XML
Extensible Markup Language (XML), Version 1.0 (second edition)
Published: 2000-10
XML_SCHEMA2
XML Schema Part 2: Datatypes
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

3. Definitions

Adaptive Item

An adaptive item is an Item that adapts either its appearance, its scoring (Response Processing) or both in response to each of the candidate's Attempts. For example, an adaptive item may start by prompting the candidate with a box for free-text entry but, on receiving an unsatisfactory answer, present a simple choice Interaction instead and award fewer marks for subsequently identifying the correct response. Adaptivity allows authors to create items for use in formative situations which both help to guide candidates through a given task while also providing an Outcome that takes into consideration their path, enabling better subsequent content sequencing decisions to be made.

Adaptive Test

An Adaptive Test is a Test that varies the items presented to the candidate based on the responses given on items already completed. This specification only supports very limited adaptivity through the use of pre-conditions and branching. See Pre-conditions and Branching.

Assessment

Assessment is the process of measuring some aspect of a candidate. In the context of this specification, Assessment is carried out using Tests and the term Assessment is treated as being equivalent to an Assessment Test.

Assessment Test

An Assessment Test is an organized collection of Items that are used to determine the values of the outcomes (e.g., level of mastery) when measuring the performance of a candidate in a particular domain. An Assessment test contains all of the necessary instructions to enable the sequencing of the items and the calculation of the outcome values (e.g., the final test score).

Assessment Variable

An Assessment Variable is a variable used to maintain a value associated with an Item Session or Test Session. For example, the value of a Response given by the candidate or the value of an Outcome for an individual item or an entire test.

Assessment Delivery System

A system for the administration and delivery of assessments to candidates. See also Delivery Engine.

Attempt

An attempt (at an Item) is the process by which the Candidate interacts with an item in one or more Candidate Sessions, possibly assigning values to or updating the associated Response Variables.

Authoring System

A system used by authors for creating and editing Items and Assessments.

Base-type

A base-type is a predefined data type that defines a value set from which values for Item Variables are drawn. These values are indivisible with respect to the *runtime model* described by this specification.

Basic Item

A basic item is an Item that contains one and only one Interaction.

Candidate

A person that participates in a test, assessment, or exam by answering questions. See also the actor candidate.

Candidate Session

A period of time during which the candidate is interacting with the Item as part of an Attempt. An attempt may consist of more than one candidate session. For example, candidates that are not sure of the answer to one question may navigate to a second question in the same test and return to the first one later. When they leave the first question they terminate the candidate session but they *do not* terminate the Attempt. The attempt is simply suspended until a subsequent candidate session concludes it, triggering Response Processing and (possibly) Feedback.

Cloning Engine

A cloning engine is a system for creating multiple similar items (Item Clones) from an Item Template.

Composite Item

A composite item is an Item that contains more than one Interaction.

Container

A container is an aggregate data type that can contain multiple values of the primitive Base-types. Containers may be empty.

Delivery Engine

The process that coordinates the rendering and delivery of the Item(s) and the evaluation of the responses to produce scores and Feedback.

Feedback

Any material presented to the candidate conditionally based on the value of an Outcome Variable. See also Integrated Feedback, Modal Feedback, and Test Feedback.

Interaction

Interactions allow the candidate to interact with the item. Through an interaction, the candidate selects or constructs a response. See also the class interaction.

Integrated Feedback

Integrated feedback is the name given to Feedback that is integrated into the item's itemBody. Unlike Modal Feedback the candidate is free to update their responses while viewing integrated feedback.

Item

The smallest exchangeable assessment object within this specification. An item is more than a 'Question' in that it contains the question and instructions to be presented, the responseProcessing to be applied to the candidates response(s) and the Feedback that may be presented (including hints and solutions). In this specification items are represented by the assessmentItem class and the term *assessment item* is used interchangeably for *item*.

Item Clone

Item Clones are items created by an Item Template.

Item Fragment

An Item Fragment is part of an item managed independently of items that include it. See also Item Set.

Item Session

An item session is the accumulation of all the Attempts at a particular item made by a candidate.

Item Set

An Item Set is a set of items that share some common characteristic. For example, all items in the set may be introduced by the same Item Fragment, in which case the fragment is often referred to as the *set leader*.

Item Template

Item templates are templates that can be used for producing large numbers of similar Items. Such items are often called cloned items. Item templates can be used to produce items by a special purpose Cloning Engine or, where Delivery Engines support them, be used directly to produce a dynamically chosen clone at the start of an Item Session. Each item cloned from an item template is identical except for the values given to a set of Template Variables. An item is therefore an item template if it declares one or more template variables and contains a set of Template Processing rules for assigning them values.

Item Variable

A variable that records part of the state of an Item Session. The candidate's responses and any outcomes assigned by Response Processing are stored in item variables. Item variables are also used to define Item Templates. Item variables are a special kind of Assessment Variable.

Material

Material means all static text, image, or media objects that are intended for the user rather than being interpreted by a processing system. Interactions are not material.

Modal Feedback

Modal feedback is the name give to Feedback that is presented to the candidate on its own, as opposed to being integrated into the item's itemBody.

Multiple Response

A multiple response is a Response Variable that is a Container for multiple values all drawn from the value set defined by one of the Base-types. A multiple response is processed as an unordered list of these values. The list may be empty.

Non-adaptive Item

An non-adaptive item is an Item that does not adapt itself in response to the candidate's Attempts.

Object Bank

An object bank is a collection of objects used in assessment, including Items, Item Fragments, Tests, or parts of tests. Object banks are not represented directly in this information model. See Integration Guide for information about how to package assessment objects for interchange.

Ordered Response

An ordered response is a Response Variable that is a Container for multiple values all drawn from the value set defined by one of the Base-types. An ordered response is processed as an ordered list (sequence) of values. The list may be empty.

Outcome

The result of an Assessment Test or Item. An outcome is represented by one or more Outcome Variables.

Outcome Processing

The process by which the values of item Outcomes (or Responses) are aggregated to make test outcomes.

Outcome Variable

Outcome variables are declared by outcome declarations. Their value is set either from a default given in the declaration itself or by a response rule encountered during Response Processing (for Item outcomes) or Outcome Processing (for Test outcomes). See also the class outcomeDeclaration.

Pool

A group of related items transported together with meta-data that describes the group as a whole. A pool is a special case of an Object Bank.

Response

The data provided by the candidate through interaction with an item, or part of an item. The values of candidate responses are represented by Response Variables.

Response Processing

The process by which the values of Response Variables are judged (scored) and the values of item Outcomes are assigned.

Response Variable

Response variables are declared by response declarations and bound to Interactions in the Item body, they record the candidate's responses. See also the class responseDeclaration

Scoring Engine

The part of the assessment system that handles the scoring based on the Candidate's responses and the Response Processing rules.

Test Feedback

The name given to feedback that is presented to the candidate conditionally based on the value of test Outcomes.

Single Response

A single response is a Response Variable that can take a single value from the set of values defined by one of the Base-types.

Template Processing

A set of rules used to set the values of the Template Variables, typically involving some random process, and thereby select the specific clone to be used for an Item Session.

Template Variable

Template variables are declared by template declarations and used to record the values required to instantiate an item template. The values determine which clone from the set of similar items defined by an Item Template is being used for a given Item Session.

Test

See Assessment.

Test Fragment

A Test Fragment is part of a test managed independently of the tests that include it.

Test Report

A Test Report is a report on the Test Session.

Test Session

A test session is the interaction of a candidate with a Test and the items it contains.

Time Dependent Item

A time dependent item is an Item that records the accumulated elapsed time for the Candidate Sessions in a Response Variable that is used during Response Processing.

Time Independent Item

A time independent item is an Item that does not use the accumulated elapsed time during Response Processing. In practice, this information may be collected by a Delivery Engine and may still be reported as part of a Test Report.

4. Items

In this specification an assessment *item* encompasses the information that is presented to a candidate *and* information about how to score the item. Scoring takes place when candidate responses are transformed into outcomes by response processing rules. It is sometimes desirable to have several different items that appear the same to the candidate but which are scored differently. In this specification, these are distinct items *by definition* and must therefore have distinct identifiers. To help facilitate the exchange of items that share significant parts of their presentation this specification supports the inclusion of separately managed item fragments (see Item and Test Fragments) in the itemBody.

Class : `assessmentItem`

Attribute : `identifier [1]:` string
The principle identifier of the item. This identifier must have a corresponding entry in the item's meta-data. See Meta-data and Usage Data for more information.

Attribute : `title [1]:` string
The title of an assessmentItem is intended to enable the item to be selected in situations where the full text of the itemBody is not available, for example when a candidate is browsing a set of items to determine the order in which to attempt them. Therefore, delivery engines may reveal the title to candidates at any time but are not required to do so.

Attribute : `label [0..1]:` string256

Attribute : `lang [0..1]:` language

Attribute : `adaptive [1]:` boolean = false
Items are classified into Adaptive Items and Non-adaptive Items.

Attribute : `timeDependent [1]:` boolean

Attribute : `toolName [0..1]:` string256
The tool name attribute allows the tool creating the item to identify itself. Other processing systems may use this information to interpret the content of application specific data, such as labels on the elements of the item's itemBody.

Attribute : `toolVersion [0..1]:` string256
The tool version attribute allows the tool creating the item to identify its version. This value must only be interpreted in the context of the toolName

Contains : responseDeclaration [*]

Contains : outcomeDeclaration [*]

Contains : templateDeclaration [*]

Contains : templateProcessing [0..1]

Contains : stylesheet [0..*]

Contains : itemBody [0..1]

Contains : responseProcessing [0..1]

Contains : modalFeedback [*]

4.1. The Item Session Lifecycle

An item session is the accumulation of all the attempts at a particular *instance* of an item made by a candidate. In some types of test, the same item may be presented to the candidate multiple times (e.g., during 'drill and practice'). Each occurrence or *instance* of the item is associated with its own item session.

The following diagram illustrates the user-perceived states of the item session. Not all states will apply to every scenario, for example feedback may not be provided for an item or it may not be allowed in the context in which the item is being used. Similarly, the candidates may not be permitted to review their responses and/or examine a model solution. In practice, systems may support only a limited number of the indicated state transitions and/or support other state transitions not shown here.

For system developers, an important first step in determining which requirements apply to their system is to identify which of the user-perceived states are supported in their system and to match the state transitions indicated in the diagram to their own event model.

The discussion that follows forms part of this specification's requirements on Delivery Engines.

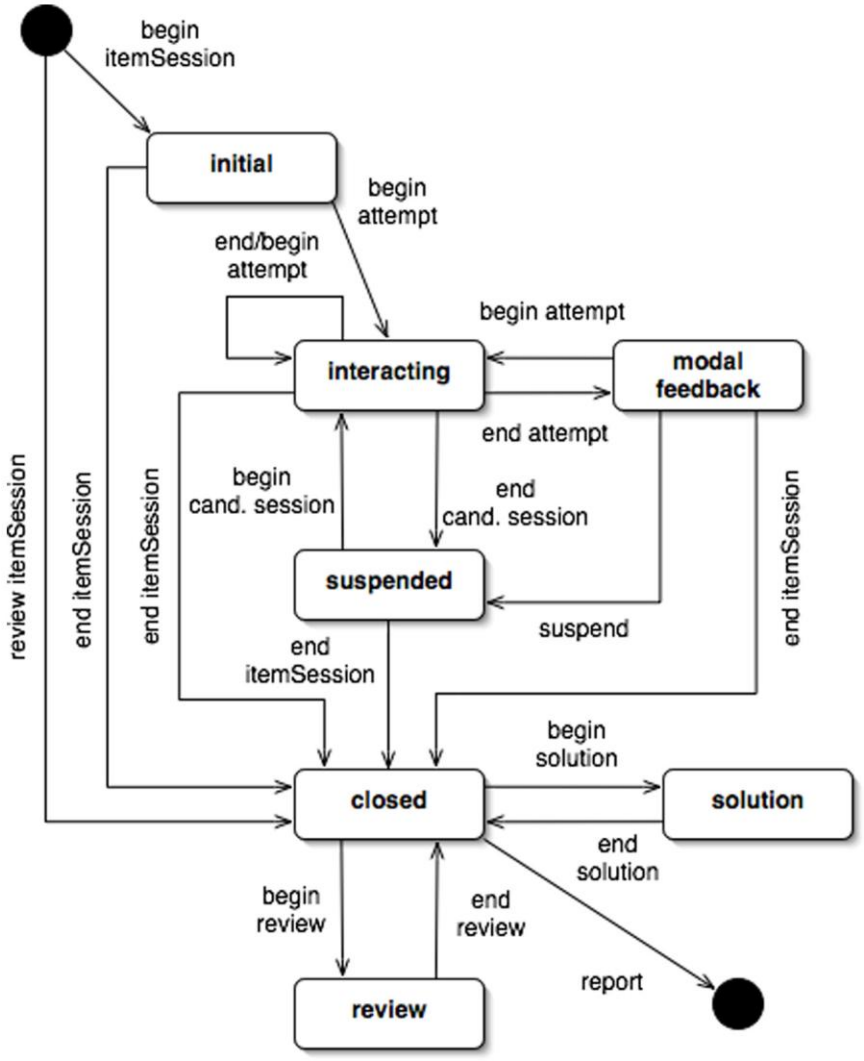

Figure 4.1 Lifecycle of an Item Session.

The session starts when the associated item first becomes eligible for delivery to the candidate. The item session's state is then maintained and updated in response to the actions of the candidate until the session is over. At any time the state of the session may be turned into an itemResult. A delivery system may also allow an itemResult to be used as the basis for a new session in order to allow a candidate's responses to be seen in the context of the item itself (and possibly compared to a solution) or even to allow a candidate to resume an interrupted session at a later time.

The initial state of an item session represents the state after it has been determined that the item will be delivered to the candidate but before the delivery has taken place.

In a typical non-Adaptive Test the items are selected in advance and the candidate's interaction with all items is reported at the end of the test session, regardless of whether or not the candidate actually attempted all the items. In effect, item sessions are created in the initial state for all

items at the start of the test and are maintained in parallel. In an Adaptive Test the items that are to be presented are selected during the session based on the responses and outcomes associated with the items presented so far. Items are selected from a large pool and the delivery engine only reports the candidate's interaction with items that have actually been selected.

A candidate's interaction with an item is broken into 0 or more attempts. During each attempt the candidate interacts with the item through one or more candidate sessions. At the end of a candidate session the item may be placed into the suspended state ready for the next candidate session. During a candidate session the *item* session is in the interacting state. Once an attempt has ended response processing takes place, after response processing a new attempt may be started.

For non-adaptive *items*, response processing typically takes place a limited number of times, usually only once. For adaptive items, no such limit is required because the response processing *adapts* the values it assigns to the outcome variables based on the path through the item. In both cases, each invocation of response processing occurs at the end of each attempt. The appearance of the item's body, and whether any modal feedback is shown, is determined by the values of the outcome variables.

When no more attempts are allowed the item session passes into the closed state. Once in the closed state the values of the response variables are fixed. A delivery system or reporting tool may still allow the item to be presented after it has reached the closed state. This type of presentation takes place in the review state, summary feedback may also be visible at this point if response processing has taken place and set a suitable outcome variable.

Finally, for systems that support the display of solutions, the item session may pass into the solution state. In this state, the candidate's responses are temporarily replaced by the correct values supplied in the corresponding responseDeclarations (or NULL if none was declared).

Class : `itemSessionControl`

Associated classes:
`testPart`, `sectionPart`

When items are referenced as part of a test, the test may impose constraints on how many attempts and which states are allowed. These constraints can be specified for individual items, for whole sections, or for an entire testPart. By default, a setting at testPart level affects all items in that part unless the setting is overridden at the assessmentSection level or ultimately at the individual assessmentItemRef. The defaults given below are used only in the absence of **any** applicable constraint.

Attribute : `maxAttempts [0..1]:` integer
For non-adaptive items, *maxAttempts* controls the maximum number of attempts allowed in the given test context. Normally this is 1 as the scoring rules for non-adaptive items are the same for each attempt. A value of 0 indicates no limit. If it is unspecified it is treated as 1 for non-adaptive

items. For adaptive items, the value of *maxAttempts* is ignored as the number of attempts is limited by the value of the *completionStatus* built-in outcome variable.

A value of maxAttempts greater than 1, by definition, indicates that any applicable feedback must be shown. This applies to both Modal Feedback and Integrated Feedback where applicable. However, once the maximum number of allowed attempts have been used (or for adaptive items, *completionStatus* has been set to *completed*) whether or not feedback is shown is controlled by the showFeedback constraint.

Attribute : `showFeedback [0..1]:` boolean
This constraint affects the visibility of feedback *after* the end of the last attempt. If it is *false* then feedback is not shown. This includes both Modal Feedback and Integrated Feedback *even if the candidate has access to the review state*. The default is *false*.

Attribute : `allowReview [0..1]:` boolean
This constraint also applies only *after* the end of the last attempt. If set to *true* the item session is allowed to enter the *review* state during which the candidate can review the itemBody along with the responses they gave, but cannot update or resubmit them. If set to *false* the candidate can **not** review the itemBody or their responses once they have submitted their last attempt. The default is *true*.

If the review state is allowed, but feedback is not, delivery systems must take extra care **not** to show integrated feedback that resulted from the last attempt as part of the review process. Feedback can however take the form of hiding material that was previously visible, as well as the more usual form of showing material that was previously hidden.

To resolve this ambiguity, for non-adaptive items the absence of feedback is defined to be the version of the itemBody displayed to the candidate at the start of each attempt. In other words, with the visibility of any integrated feedback determined by the **default** values of the outcome variables and **not** the values of the outcome variables updated by the invocation of response processing.

For Adaptive Items the situation is complicated by the iterative nature of response processing which makes it hard to identify the appropriate state in which to place the item for review. To avoid requiring delivery engines to cache the values of the outcome variables the setting of showFeedback should be ignored for adaptive items when allowReview is *true*. When in the review state, the *final* values of the outcome variables should be used to determine the visibility of integrated feedback.

Attribute : `showSolution [0..1]:` boolean
This constraint controls whether or not the system may provide the candidate with a way of entering the *solution* state. The default is *false*.

Attribute : `allowComment [0..1]:` boolean
Some delivery systems support the capture of candidate comments. The comment is not part of the assessed responses but provides feedback from the candidate to the other actors in the

assessment process. This constraint controls whether or not the candidate is allowed to provide a comment on the item during the session.

Attribute : `allowSkipping [0..1]:` boolean = true
An item is defined to be skipped if the candidate has not provided any response. In other words, *all* response variables are submitted with their default value or are NULL. This definition is consistent with the numberResponded operator available in outcomeProcessing. If *false*, candidates are not allowed to skip the item, or in other words, they are not allowed to submit the item until they have provided a non-default value for at least one of the response variables. By definition, an item with no response variables cannot be skipped. The value of this attribute is only applicable when the item is in a testPart with individual submission mode. Note that if allowSkipping is *true* delivery engines must ensure that the candidate can choose to submit no response, for example, through the provision of a "skip" button.

Attribute : `validateResponses [0..1]:` boolean
This attribute controls the behavior of delivery engines when the candidate submits an *invalid* response. An invalid response is defined to be a response which does not satisfy the constraints imposed by the interaction with which it is associated. See interaction for more information. When validateResponses is turned on (*true*) then the candidates are not allowed to submit the item until they have provided valid responses for all interactions. When turned off (*false*) invalid responses may be accepted by the system. The value of this attribute is only applicable when the item is in a testPart with individual submission mode. (See Navigation and Submission.)

5. Item Variables

Abstract class : `variableDeclaration`

Derived classes:
`outcomeDeclaration`, `responseDeclaration`, `templateDeclaration`

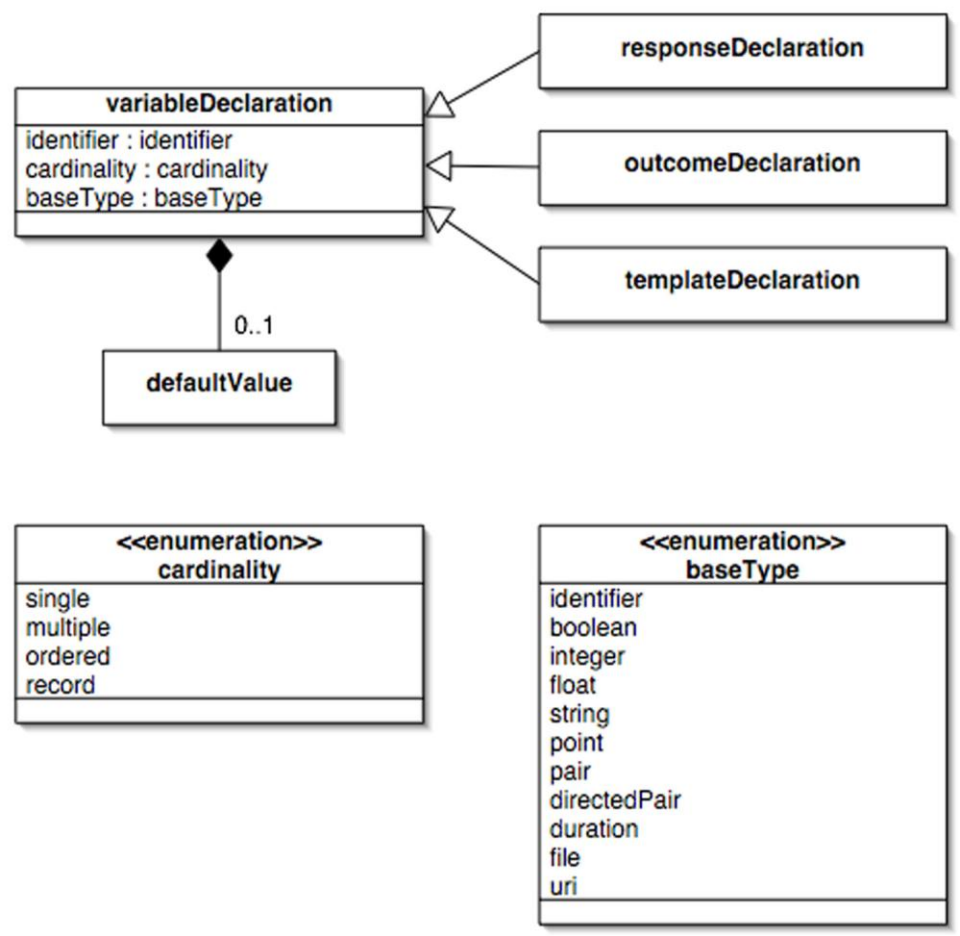

Figure 5.1 Variable Declarations.

Item variables are declared by variable declarations. All variables must be declared except for the built-in session variables referred to below which are declared implicitly and must not be declared. The purpose of the declaration is to associate an identifier with the variable and to identify the runtime type of the variable's value.

An item variable may have no value at all, in which case it is said to have the special value NULL. For example, if the candidate has not yet had an opportunity to respond to an interaction then any associated response variable will have a NULL value. Empty containers and empty strings are always treated as NULL values.

Attribute : `identifier [1]:` identifier
The identifiers of the built-in session variables are reserved. They are *completionStatus*, *numAttempts* and *duration*. All item variables declared in an item share the same namespace. Different items have different namespaces.

Attribute : `cardinality [1]:` cardinality
Each variable is either single valued or multi-valued. Multi-valued variables are referred to as containers and come in ordered, unordered and record types. See cardinality for more information.

Attribute : `baseType [0..1]:` baseType
The value space from which the variable's value can be drawn (or in the case of containers, from which the individual values are drawn) is identified with a baseType. The baseType selects one of a small set of predefined types that are considered to have atomic values within the runtime data model. Variables with record cardinality have no base-type.

Contains : defaultValue [0..1]
An optional default value for the variable. The point at which a variable is set to its default value varies depending on the type of item variable.

Class : `value`

Associated classes:
`ordinaryStatistic`, `templateVariable`, `candidateResponse`, `correctResponse`, `outcomeVariable`, `defaultValue`

A class that can represent a single value of any baseType in variable declarations and result reports. The base-type is defined by the baseType attribute of the declaration except in the case of variables with record cardinality.

Attribute : `fieldIdentifier [0..1]:` identifier
This attribute is *only* used for specifying the field identifier for a value that forms part of a record.

Attribute : `baseType [0..1]:` baseType
This attribute is *only* used for specifying the base-type of a value that forms part of a record.

Class : `defaultValue`

Associated classes:
`variableDeclaration`

Attribute : `interpretation [0..1]:` string
A human readable interpretation of the default value.

Contains : value [1..*]

Enumeration: `cardinality`

`single`

`multiple`

`ordered`

`record`

An expression or itemVariable can either be single-valued or multi-valued. A multi-valued expression (or variable) is called a container. A container contains a list of values, this list may be empty in which case it is treated as NULL. All the values in a multiple or ordered container are drawn from the same value set, however, containers may contain multiple occurrences of the *same* value. In other words, [A,B,B,C] is an acceptable value for a container. A container with cardinality multiple and value [A,B,C] is equivalent to a similar one with value [C,B,A] whereas these two values would be considered distinct for containers with cardinality ordered. When used as the value of a response variable this distinction is typified by the difference between selecting choices in a multi-response multi-choice task and ranking choices in an order objects task. In the language of [ISO11404] a container with multiple cardinality is a "bag-type", a container with ordered cardinality is a "sequence-type" and a container with record cardinality is a "record-type".

The record container type is a special container that contains a set of independent values each identified by its own identifier and having its own base-type. This specification does not make use of the record type directly however it is provided to enable customInteractions to manipulate more complex responses and customOperators to return more complex values.

Enumeration: `baseType`

A base-type is simply a description of a set of atomic values (atomic to this specification). Note that several of the baseTypes used to define the runtime data model have identical definitions to those of the basic data types used to define the values for attributes in the specification itself. The use of an enumeration to define the set of baseTypes used in the *runtime* model, as opposed to the use of classes with similar names, is designed to help distinguish between these two distinct levels of modelling.

`identifier`
The set of identifier values is the same as the set of values defined by the identifier class.

`boolean`
The set of boolean values is the same as the set of values defined by the boolean class.

`integer`
The set of integer values is the same as the set of values defined by the integer class.

`float`
The set of float values is the same as the set of values defined by the float class.

`string`
The set of string values is the same as the set of values defined by the string class.

`point`
A point value represents an integer tuple corresponding to a graphic point. The two integers correspond to the horizontal (x-axis) and vertical (y-axis) positions respectively. The up/down and left/right senses of the axes are context dependent.

`pair`
A pair value represents a pair of identifiers corresponding to an association between two objects. The association is undirected so (A,B) and (B,A) are equivalent.

`directedPair`
A directedPair value represents a pair of identifiers corresponding to a directed association between two objects. The two identifiers correspond to the source and destination objects.

`duration`
A duration value specifies a distance (in time) between two time points. In other words, a time period as defined by [ISO8601]. Durations are measured in seconds and may have a fractional part.

`file`
A file value is any sequence of octets (bytes) qualified by a content-type and an optional filename given to the file (for example, by the candidate when uploading it as part of an interaction). The content type of the file is one of the MIME types defined by [RFC2045].

`uri`
A URI value is a Uniform Resource Identifier as defined by [URI].

Class : `mapping`

Associated classes:
`responseDeclaration`, `categorizedStatistic`

A special class used to create a mapping from a source set of any baseType (except file and duration) to a single float. Note that mappings from values of base type float should be avoided due to the difficulty of matching floating point values, see the match operator for more details. When mapping containers the result is the *sum* of the mapped values from the target set. See mapResponse for details.

Attribute : `lowerBound [0..1]`: float
The lower bound for the result of mapping a container. If unspecified there is no lower-bound.

Attribute : `upperBound [0..1]`: float
The upper bound for the result of mapping a container. If unspecified there is no upper-bound.

Attribute : `defaultValue [1]:` float = 0
The default value from the target set to be used when no explicit mapping for a source value is given.

Contains : mapEntry [1..*]
The map is defined by a set of mapEntries, each of which maps a single value from the source set onto a single float.

Class : `mapEntry`

Associated classes:
`mapping`

Attribute : `mapKey [1]:` valueType
The source value

Attribute : `mappedValue [1]:` float
The mapped value

5.1. Response Variables

Class : `responseDeclaration (`variableDeclaration)

Associated classes:
`assessmentItem`

Response variables are declared by response declarations and bound to interactions in the itemBody.

At runtime, response variables are instantiated as part of an item session. Their values are always initialized to NULL (no value) regardless of whether or not a default value is given in the declaration. A response variable with a NULL value indicates that the candidate has not offered a response, either because they have not attempted the item at all or because they have attempted it and chosen not to provide a response.

If a default value has been provided for a response variable then the variable is set to this value at the start of the first attempt. If the candidate never attempts the item, in other words, the item session passes straight from the initial state to the closed state without going through the interacting state, then the response variable remains NULL and the default value is never used.

Implementors of Delivery Engines should take care when implementing user interfaces for items with default response variable values. If the associated interaction is left in the default state (i.e., representing the default value) then it is important that the system is confident that the candidate intended to submit this value and has not simply failed to notice that a default has been provided. This is especially true if the candidate's attempt ended due to some external event, such as

running out of time. The techniques required to distinguish between these cases are an issue for user interface design and are therefore out of scope for this specification.

Contains : correctResponse [0..1]
A response declaration may assign an optional correctResponse. This value may indicate the only possible value of the response variable to be considered correct or merely just *a* correct value. For responses that are being measured against a more complex scale than correct/incorrect this value should be set to the (or an) optimal value. Finally, for responses for which no such optimal value is defined the correctResponse must be omitted. If a delivery system supports the display of a solution then it should display the correct values of responses (where defined) to the candidate. When correct values are displayed they must be clearly distinguished from the candidate's own responses (which may be hidden completely if necessary).

Contains : mapping [0..1]
The mapping provides a mapping from the set of base values to a set of numeric values for the purposes of response processing. See mapResponse for information on how to use the mapping.

Contains : areaMapping [0..1]
The areaMapping, which may only be present in declarations of variables with baseType point, provides an alternative form of mapping which tests against areas of the coordinate space instead of mapping single values (i.e., single points).

Class : `correctResponse`

Associated classes:
`responseDeclaration`, `responseVariable`

Attribute : `interpretation [0..1]`: string
A human readable interpretation of the correct value.

Contains : value [1..*]

Class : `areaMapping`

Associated classes:
`responseDeclaration`

A special class used to create a mapping from a source set of point values to a target set of float values. When mapping containers, the result is the *sum* of the mapped values from the target set. See mapResponsePoint for details. The attributes have the same meaning as the similarly named attributes on mapping.

Attribute : `lowerBound [0..1]`: float

Attribute : `upperBound [0..1]`: float

Attribute : `defaultValue [1]:` float = 0

Contains : areaMapEntry [1..*] {ordered}
The map is defined by a set of areaMapEntries, each of which maps an area of the coordinate space onto a single float. When mapping points each area is tested in turn, with those listed first taking priority in the case where areas overlap and a point falls in the intersection.

Class : `areaMapEntry`

Associated classes:
`areaMapping`

Attribute : `shape [1]:` shape
The shape of the area.

Attribute : `coords [1]:` coords
The size and position of the area, interpreted in conjunction with the shape.

Attribute : `mappedValue [1]:` float
The mapped value

5.1.1. Built-in Response Variables

There are two built-in response variables, *numAttempts* and *duration* that are declared implicitly and must not appear in a responseDeclaration.

All Delivery Engines must maintain the value of *numAttempts*. This is a single integer that records the number of attempts at each item the candidate has taken. The value defaults to 0 initially and then increases by 1 at the *start* of each attempt.

Systems that support Time Dependent Items must also maintain the value of *duration*. The duration is defined as being a single float that records the accumulated time (in seconds) of all Candidate Sessions for all Attempts. In other words the time between the beginning and the end of the item session **minus** any time the session was in the suspended state. The resolution of the duration must be at least 1s and should be 0.1s or smaller. If the resolution is denoted by *epsilon* then each value of duration represents the range of values duration<=t<duration+*epsilon*. In other words, duration values are truncated.

The value of duration for items that are not time dependent must not be used in any item or test-level expression, though systems should still report it in itemResults when it is known.

5.2. Outcome Variables

Class : `outcomeDeclaration` (variableDeclaration)

Associated classes:

`assessmentTest, assessmentItem`

Outcome variables are declared by outcome declarations. Their value is set either from a default given in the declaration itself or by a responseRule during responseProcessing.

Items that declare a numeric outcome variable representing the candidate's overall performance on the item should use the outcome name 'SCORE' for the variable.

At runtime, outcome variables are instantiated as part of an item session. Their values may be initialized with a default value and/or set during responseProcessing. If no default value is given in the declaration then the outcome variable is initialized to NULL *unless* the outcome is of a numeric type (integer or float) in which case it is initialized to 0.

For Non-adaptive Items, the values of the outcome variables are reset to their default values prior to each invocation of responseProcessing. For Adaptive Items the outcome variables *retain* the values that were assigned to them during the previous invocation of response processing. For more information, see Response Processing.

Attribute : `view [*]:` view
The intended audience for an outcome variable can be set with the view attribute. If no view is specified the outcome is treated as relevant to *all* views. Complex items, such as adaptive items or complex templates, may declare outcomes that are of no interest to the candidate at all, but are merely used to hold intermediate values or other information useful during the item or test session. Such variables should be declared with a view of author (for item outcomes) or testConstructor (for test outcomes). Systems may exclude outcomes from result reports on the basis of their declared view if appropriate.

Attribute : `interpretation [0..1]:` string
A human interpretation of the variable's value.

Attribute : `longInterpretation [0..1]:` uri
An optional link to an extended interpretation of the outcome variable's value.

Declared outcomes with numeric types should indicate their range of possible values using normalMaximum and normalMinimum, especially if this range differs from [0,1].

Attribute : `normalMaximum [0..1]:` float
The normalMaximum attribute optionally defines the maximum *magnitude* of numeric outcome variables, it must be a positive value. If given, the outcome's value can be divided by normalMaximum and then truncated (if necessary) to obtain a normalized score in the range [-1.0,1.0]. normalMaximum has no affect on responseProcessing or the values that the outcome variable itself can take.

Attribute : `normalMinimum [0..1]:` float
The normalMinimum attribute optionally defines the minimum value of numeric outcome variables, it may be negative.

Attribute : `masteryValue [0..1]:` float
The masteryValue attribute optionally defines a value for numeric outcome variables above which the aspect being measured is considered to have been mastered by the candidate.

Contains : lookupTable [0..1]

Abstract class : `lookupTable`

Derived classes:
`interpolationTable`, `matchTable`
Associated classes:
`outcomeDeclaration`

An abstract class associated with an outcomeDeclaration used to create a lookup table from a numeric source value to a single outcome value in the declared value set. A lookup table works in the reverse sense to the similar mapping as it defines how a source numeric value is transformed into the outcome value, whereas a (response) mapping defines how the response value is mapped onto a target numeric value.

The transformation takes place using the lookupOutcomeValue rule within responseProcessing or outcomeProcessing.

Attribute : `defaultValue [0..1]:` valueType
The default outcome value to be used when no matching table entry is found. If omitted, the NULL value is used.

Class : `matchTable` (lookupTable)

A matchTable transforms a source integer by finding the first matchTableEntry with an exact match to the source.

Contains : matchTableEntry [1..*]

Class : `matchTableEntry`

Associated classes:
`matchTable`

Attribute : `sourceValue [1]:` integer
The source integer that must be matched exactly.

Attribute : `targetValue [1]:` valueType
The target value that is used to set the outcome when a match is found.

Class : `interpolationTable` (lookupTable)

An interpolationTable transforms a source float (or integer) by finding the first interpolationTableEntry with a sourceValue that is less than or equal to (subject to includeBoundary) the source value.

For example, an interpolation table can be used to map a raw numeric score onto an identifier representing a grade. It may also be used to implement numeric transformations such as those from a simple raw score to a value on a calibrated scale.

Contains : interpolationTableEntry [1..*]

Class : `interpolationTableEntry`

Associated classes:
`interpolationTable`

Attribute : `sourceValue [1]:` float
The lower bound for the source value to match this entry.

Attribute : `includeBoundary [0..1]:` boolean = true
Determines if an exact match of sourceValue matches this entry. If *true*, the default, then an exact match of the value is considered a match of this entry.

Attribute : `targetValue [1]:` valueType
The target value that is used to set the outcome when a match is found.

5.2.1. Built-in Outcome Variables

There is one built-in outcome variable, *completionStatus*, that is declared implicitly and must not appear in an outcomeDeclaration.

Delivery Engines must maintain the value of the built-in outcome variable *completionStatus*, a single identifier. It starts with the reserved value "not_attempted". At the start of the first attempt it changes to the reserved value "unknown". It remains with this value for the duration of the item session unless set to a different value by a setOutcomeValue rule in responseProcessing. There are four permitted values: *completed*, *incomplete*, *not_attempted*, and *unknown*. Any one of these values may be set during response processing, for definitions of the meanings see [CMI]. If an Adaptive Item sets completionStatus to *completed* then the session must be placed into the closed state, however an item session is **not** required to wait for the completed signal before terminating, it may terminate in response to a direct request from the candidate, through running out of time or through some other exceptional circumstance. Adaptive Items *must* maintain a suitable value and should set completionStatus to "completed" to indicate when the cycle of interaction, response processing, and feedback must stop. Non-adaptive Items are not *required* to set a value for completionStatus, but they may do so. Delivery Engines are encouraged to use the value of completionStatus when communicating using [CMI]. See the accompanying integration guide for more details.

6. Content Model

Class : `itemBody (`bodyElement**)**

Associated classes:
`assessmentItem`

Contains : block [*]

The item body contains the text, graphics, media objects, and interactions that describe the item's content and information about how it is structured. The body is presented by combining it with stylesheet information, either explicitly or implicitly using the default style rules of the delivery or authoring system.

The body must be presented to the candidate when the associated item session is in the interacting state. In this state, the candidate must be able to interact with each of the visible interactions and therefore set or update the values of the associated response variables. The body may be presented to the candidate when the item session is in the closed or review state. In these states, although the candidate's responses should be visible, the interactions must be disabled so as to prevent the candidate from setting or updating the values of the associated response variables. Finally, the body may be presented to the candidate in the solution state, in which case the correct values of the response variables must be visible and the associated interactions disabled.

The content model employed by this specification uses many concepts taken directly from [XHTML]. In effect, this part of the specification defines a profile of XHTML. Only some of the elements defined in XHTML are allowable in an assessmentItem and of those that are, some have additional constraints placed on their attributes. Only those elements from XHTML that are explicitly defined within this specification can be used. See XHTML Elements for details. Finally, this specification defines some new elements which are used to represent the interactions and to control the display of Integrated Feedback and content restricted to one or more of the defined content views.

Abstract class : `bodyElement`

Derived classes:
`atomicBlock`, `atomicInline`, `caption`, `choice`, `col`, `colgroup`, `div`, `dl`, `dlElement`, `hr`, `interaction`, `itemBody`, `li`, `object`, `ol`, `printedVariable`, `prompt`, `simpleBlock`, `simpleInline`, `table`, `tableCell`, `tbody`, `templateElement`, `tfoot`, `thead`, `tr`, `ul`

The root class of all content objects in the item content model is the bodyElement. It defines a number of attributes that are common to all elements of the content model.

Attribute : `id [0..1]:` identifier
The id of a body element must be unique within the item.

Attribute : `class [*]:` styleclass
Classes can be assigned to individual body elements. Multiple class names can be given. These class names identify the element as being a member of the listed classes. Membership of a class can be used by authoring systems to distinguish between content objects that are not differentiated by this specification. Typically, this information is used to apply different formatting based on definitions in an associated stylesheet.

Attribute : `lang [0..1]:` language
The main language of the element. This attribute is optional and will usually be inherited from the enclosing element.

Attribute : `label [0..1]:` string256
The label attribute provides authoring systems with a mechanism for labeling elements of the content model with application specific data. If an item uses labels then values for the associated toolName and toolVersion attributes must also be provided.

6.1. Basic Classes

Underpinning the content model are a number of abstract classes that are used to group elements of the body into categories that define peer-groups.

Abstract class : `objectFlow`

Derived classes:
`flow`, `param`
Associated classes:
`object`

Elements that can appear within an object.

Abstract class : `inline`

Derived classes:
`inlineInteraction`, `inlineStatic`
Associated classes:
`simpleInline`, `dt`, `caption`, `atomicBlock`

Elements that behave as spans of text, such as the contents of paragraphs.

Abstract class : `block`

Derived classes:
`blockInteraction`, `blockStatic`, `customInteraction`, `positionObjectStage`
Associated classes:
`itemBody`, `simpleBlock`

Elements that provide structure to the text, such as paragraphs, tables etc. Most elements are either inline or block elements.

Abstract class : `flow` (objectFlow)

Derived classes:
`blockInteraction`, `customInteraction`, `flowStatic`, `include`, `inlineInteraction`
Associated classes:
`tableCell`, `div`, `dd`, `li`

Elements that can appear inside list items, table cells, etc. which includes block-type and inline-type elements.

Attribute : `base [0..1]`: uri
An optional URI used to change the base for resolving relative URI for the scope of this object. Particular care needs to be taken when resolving relative URI included as part of an Item Fragment. See Item and Test Fragments for more information.

Abstract class : `inlineStatic` (inline)

Derived classes:
`atomicInline`, `gap`, `hottext`, `math`, `object`, `printedVariable`, `simpleInline`, `templateInline`, `textRun`
Associated classes:
`hottext`, `prompt`, `templateInline`

A sub-class of inline that excludes interactions.

Abstract class : `blockStatic` (block)

Derived classes:
`atomicBlock`, `div`, `dl`, `hr`, `math`, `ol`, `simpleBlock`, `table`, `templateBlock`, `ul`
Associated classes:
`templateBlock`, `gapMatchInteraction`, `hottextInteraction`

A sub-class of block that excludes interactions.

Abstract class : `flowStatic` (flow)

Derived classes:
`atomicBlock`, `atomicInline`, `div`, `dl`, `hottext`, `hr`, `math`, `object`, `ol`, `printedVariable`, `simpleBlock`, `simpleInline`, `table`, `templateBlock`, `templateInline`, `textRun`, `ul`
Associated classes:
`simpleAssociableChoice`, `testFeedback`, `modalFeedback`, `simpleChoice`

A sub-class of flow that excludes interactions.

The following classes define a small number of common element types used by XHTML.

Abstract class : `simpleInline` (bodyElement, flowStatic, inlineStatic)

Derived classes:
`a`, `abbr`, `acronym`, `b`, `big`, `cite`, `code`, `dfn`, `em`, `feedbackInline`, `i`, `kbd`, `q`, `samp`, `small`, `span`, `strong`, `sub`, `sup`, `tt`, `var`

Contains : inline [*]

Abstract class : `simpleBlock` (blockStatic, bodyElement, flowStatic)

Derived classes:
`blockquote`, `feedbackBlock`, `rubricBlock`

Contains : block [*]

Abstract class : `atomicInline` (bodyElement, flowStatic, inlineStatic)

Derived classes:
`br`, `img`

Abstract class : `atomicBlock` (blockStatic, bodyElement, flowStatic)

Derived classes:
`address`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `p`, `pre`

Contains : inline [*]

Class : `textRun` (flowStatic, inlineStatic, textOrVariable)

A text run is simply a run of characters. Unlike all other elements in the content model it is not a sub-class of bodyElement. To assign attributes to a run of text you must use the span element instead.

6.2. XHTML Elements

The structural elements of the content model that are taken from [XHTML] are documented in groups according to their suggested classification in [XHTML_MOD]. Only those attributes listed here may be used (including attributes inherited from parent classes). By default, elements and attributes have the same interpretation and restrictions as the corresponding elements and attributes in [XHTML].

6.2.1. Text Elements

Class : `abbr` (simpleInline)

Note that the title attribute defined by XHTML is not supported.

Class : `acronym` (simpleInline)

Note that the title attribute defined by XHTML is not supported.

Class : `address` (atomicBlock)

Class : `blockquote` (simpleBlock)

Attribute : `cite [0..1]:` uri

Class : `br` (atomicInline)

Class : `cite` (simpleInline)

Class : `code` (simpleInline)

Class : `dfn` (simpleInline)

Class : `div` (blockStatic, bodyElement, flowStatic)

Contains : flow [*]

Class : `em` (simpleInline)

Class : `h1` (atomicBlock)

Class : `h2` (atomicBlock)

Class : `h3` (atomicBlock)

Class : `h4` (atomicBlock)

Class : `h5` (atomicBlock)

Class : `h6` (atomicBlock)

Class : `kbd` (simpleInline)

Class : `p` (atomicBlock)

Class : `pre` (atomicBlock)

Although pre inherits from atomicBlock it must not contain, either directly or indirectly, any of the following objects: img, object, big, small, sub, sup.

Class : `q` (simpleInline)

Attribute : `cite [0..1]:` uri

Class : `samp` (simpleInline)

Class : `span` (simpleInline)

Class : `strong` (simpleInline)

Class : `var` (simpleInline)

6.2.2. List Elements

Class : `dl` (blockStatic, bodyElement, flowStatic)

Contains : dlElement [*]

Abstract class : `dlElement` (bodyElement)

Derived classes:
`dd`, `dt`
Associated classes:
`dl`

Class : `dt` (dlElement)

Contains : inline [*]

Class : `dd` (dlElement)

Contains : flow [*]

Class : `ol` (blockStatic, bodyElement, flowStatic)

Contains : li [*]

Class : `ul` (blockStatic, bodyElement, flowStatic)

Contains : li [*]

Class : `li` (bodyElement)

Associated classes:
ul, ol

Contains : flow [*]

6.2.3. Object Elements

Class : object (bodyElement, flowStatic, inlineStatic)

Associated classes:
positionObjectStage, graphicInteraction, positionObjectInteraction, mediaInteraction, drawingInteraction, gapImg

Contains : objectFlow [*]

Attribute : data [1]: string
The data attribute provides a URI for locating the data associated with the object.

Attribute : type [1]: mimeType

Attribute : width [0..1]: length

Attribute : height [0..1]: length

Class : param (objectFlow)

Attribute : name [1]: string
The name of the parameter, as interpreted by the object.

Attribute : value [1]: string
The value to pass to the object for the named parameter. This value is subject to template variable expansion. If the value is the name of a template variable that was declared with the paramVariable set to *true* then the template variable's *value* is passed to the object as the value for the given parameter.

When expanding a template variable as a parameter value, types other than identifiers, strings and uris must be converted to strings. Numeric types are converted to strings using the "%i" or "%G" formats as appropriate (see printedVariable for a discussion of numeric formatting). Values of base-type boolean are expanded to one of the strings "true" or "false". Values of base-type point are expanded to two space-separated integers in the order horizontal coordinate, vertical coordinate, using "%i" format. Values of base-type pair and directedPair are converted to a string consisting of the two identifiers, space separated. Values of base-type duration are converted using "%G" format. Values of base-type file cannot be used in parameter expansion.

If the valuetype is *REF* the template variable must be of base-type uri.

Attribute : `valuetype [1]:` paramType = DATA
This specification supports the use of *DATA* and *REF* but **not** *OBJECT*.

Attribute : `type [0..1]:` mimeType
Used to provide a type for values valuetype REF.

Enumeration: `paramType`

```
DATA

REF
```

6.2.4. Presentation Elements

Class : `b` (simpleInline)

Class : `big` (simpleInline)

Class : `hr` (blockStatic, bodyElement, flowStatic)

Class : `i` (simpleInline)

Class : `small` (simpleInline)

Class : `sub` (simpleInline)

Class : `sup` (simpleInline)

Class : `tt` (simpleInline)

6.2.5. Table Elements

Class : `caption` (bodyElement)

Associated classes:
`table`

Contains : inline [*]

Class : `col` (bodyElement)

Associated classes:
`table`, `colgroup`

Attribute : `span [0..1]:` integer = 1

Class : `colgroup` (bodyElement)

Associated classes:
table

Attribute : span [0..1]: integer = 1

Contains : col [*]

Class : table (blockStatic, bodyElement, flowStatic)

Attribute : summary [0..1]: string

Contains : caption [0..1]

Contains : col [*]
If a table *directly* contains a col then it must not contain any colgroup elements.

Contains : colgroup [*]
If a table contains a colgroup it must not *directly* contain any col elements.

Contains : thead [0..1]

Contains : tfoot [0..1]

Contains : tbody [1..*]

Abstract class : tableCell (bodyElement)

Derived classes:
td, th
Associated classes:
tr

In XHTML, table cells are represented by either th or td and these share the following attributes and content model:

Attribute : headers [*]: identifier

Attribute : scope [0..1]: tableCellScope

Attribute : abbr [0..1]: string

Attribute : axis [0..1]: string

Attribute : rowspan [0..1]: integer

Attribute : colspan [0..1]: integer

Contains : flow [*]

Enumeration: `tableCellScope`

```
row

col

rowgroup

colgroup
```

Class : `tbody` (bodyElement)

Associated classes:
`table`

Contains : tr [1..*]

Class : `td` (tableCell)

Class : `tfoot` (bodyElement)

Associated classes:
`table`

Contains : tr [1..*]

Class : `th` (tableCell)

Class : `thead` (bodyElement)

Associated classes:
`table`

Contains : tr [1..*]

Class : `tr` (bodyElement)

Associated classes:
`thead`, `tbody`, `tfoot`

Contains : tableCell [1..*]

6.2.6. Image Element

Class : `img` (atomicInline)

Attribute : `src [1]:` uri

Attribute : `alt [1]:` string256

Attribute : `longdesc [0..1]:` uri

Attribute : `height [0..1]:` length

Attribute : `width [0..1]:` length

6.2.7. Hypertext Element

Class : `a (`simpleInline**)**

Although a inherits from simpleInline it must not contain, either directly or indirectly, another a.

Attribute : `href [1]:` uri

Attribute : `type [0..1]:` mimeType

6.3. MathML

[MathML] defines a Markup Language for describing mathematical notation using XML. The primary purpose of MathML is to provide a language for embedding mathematical expressions into other documents, in particular into HTML documents.

Class : `math (`blockStatic, flowStatic, inlineStatic**)**

The math class is defined externally by the MathML specification. It can behave in the item's content model as an inline, block or flow element.

6.3.1. Combining Template Variables and MathML

It is often desirable to vary elements of a mathematical expression when creating item templates. Although it is impossible to embed objects such as printedVariable defined for that purpose within a math object the techniques described in this section can be used to achieve a similar effect.

In MathML, numbers are represented either by the <mn> or <cn> elements, for presentation or content representation respectively. Similarly, <mi> and <ci> represent identifiers. If mathVariable is set in a template variable's declaration then all instances of <mi> and <ci> that match the name of the template variable must be replaced by <mn> and <cn> respectively with the template variable's value as their content.

It is possible that this technique of expanding template variables will be extended to other elements of MathML in future.

6.4. Variable Content

This specification defines two methods by which the content of an assessmentItem can vary depending on the state of the item session.

The first method is based on the value of an outcome variable.

Abstract class : `feedbackElement`

Derived classes:
`feedbackBlock`, `feedbackInline`

Attribute : `outcomeIdentifier [1]:` identifier
The identifier of an outcome variable that must have a base-type of identifier and be of either single or multiple cardinality. The visibility of the feedbackElement is controlled by assigning a value (or values) to this outcome variable during responseProcessing.

Attribute : `showHide [1]:` showHide = show
The showHide attribute determines how the visibility of the feedbackElement is controlled. If set to show then the feedback is hidden by default and shown only if the associated outcome variable matches, or contains, the value of the identifier attribute. If set to hide then the feedback is shown by default and hidden if the associated outcome variable matches, or contains, the value of the identifier attribute.

Attribute : `identifier [1]:` identifier
The identifier that determines the visibility of the feedback in conjunction with the showHide attribute.

A feedback element that forms part of a Non-adaptive Item must not contain an interaction object, either directly or indirectly.

When an interaction is contained in a hidden feedback element it must also be hidden. The candidate must not be able to set or update the value of the associated response variables.

Enumeration: `showHide`

`show`

`hide`

Class : `feedbackBlock` (feedbackElement, simpleBlock)

Class : `feedbackInline` (feedbackElement, simpleInline)

Class : `rubricBlock` (simpleBlock)

Associated classes:

assessmentSection

Attribute : `view [1..*]:` view
The views in which the rubric block's content are to be shown.

A rubric block identifies part of an assessmentItem's itemBody that represents instructions to one or more of the actors that view the item. Although rubric blocks are defined as simpleBlocks they must not contain interactions.

The visibility of nested bodyElements or rubricBlocks is determined by the outermost element. In other words, if an element is determined to be hidden then all of its content is hidden including conditionally visible elements for which the conditions are satisfied and that therefore would otherwise be visible.

Class : `printedVariable (`bodyElement, flowStatic, inlineStatic, textOrVariable)

Attribute : `identifier [1]:` identifier
The outcome variable or template variable that must have been defined and have single cardinality. The values of response variables cannot be printed directly as their values are implicitly known to the candidate through the interactions they are bound to. If necessary, their values can be assigned to outcomes during responseProcessing and displayed to the candidate as part of a bodyElement visible only in the appropriate feedback states.

If the variable's value is NULL then the element is ignored.

Variables of baseType string are treated as simple runs of text.

Variables of baseType integer or float are converted to runs of text (strings) using the formatting rules described below. Float values should only be formatted in the e, E, f, g, G, r or R styles.

Variables of baseType duration are treated as floats, representing the duration in seconds.

Attribute : `format [0..1]:` string256
The format conversion specifier to use when converting numerical values to strings. See Number Formatting Rules for details.

Attribute : `base [0..1]:` integer = 10
The number base to use when converting integer variables to strings with the *i* conversion type code.

Variables of baseType file are rendered using a control that enables the user to open the file. The control should display the name associated with the file, if any.

Variables of baseType uri are rendered using a control that enables the user to open the identified resource, for example, by following a hypertext link in the case of a URL.

Abstract class : `textOrVariable`

Derived classes:
`printedVariable`, `textRun`
Associated classes:
`gapText`

An abstract class used to enable variable value substitution in contexts where only plain text is allowed, such as the text of an inlineChoice.

6.4.1. Number Formatting Rules

The syntax of the format attribute is based on the format conversion specifiers defined in the C programming language [ISO_9899] for use with *printf* and related functions.

Each conversion specifier starts with a '%' character and is followed by zero or more flag characters (#, 0, -, " " *[space]* and +), an optional digit string indicating the minimum field width, an optional precision (consisting of a "." followed by zero or more digits) and finally one of the conversion type codes: E, e, f, G, g, r, R, i, o, X, or x. These are interpreted according to the C standard with the exception of *i*, which may be used to format numbers in bases other than 10 using the base attribute, and *r*/*R* which round to the number of significant figures given by the precision in the same way as *g*/*G* except that scientific format is *only* used if the requested number of significant figures is less than the number of digits to the left of the decimal point.

6.5. Formatting Items with Stylesheets

Class : `stylesheet`

Associated classes:
`assessmentItem`

Used to associate an external stylesheet with an assessmentItem.

Attribute : `href [1]`: uri
The identifier or location of the external stylesheet.

Attribute : `type [1]`: mimeType
The type of the external stylesheet.

Attribute : `media [0..1]`: string
An optional media descriptor that describes the media to which this stylesheet applies.

Attribute : `title [0..1]`: string
An optional title for the stylesheet.

7. Interactions

Abstract class : `interaction` (bodyElement)

Derived classes:
`blockInteraction`, `customInteraction`, `inlineInteraction`, `positionObjectInteraction`

Interactions allow the candidate to *interact* with the item. Through an interaction, the candidate selects or constructs a response. The candidate's responses are stored in the response variables. Each interaction is associated with (at least) one response variable.

Attribute : `responseIdentifier [1]`: identifier
The response variable associated with the interaction.

The state of the interaction is used to set the value of the associated response variable. The declaration of the associated response variable constrains the value of the response to be of a particular baseType and cardinality. Some interactions impose additional constraints on the set of allowable responses, for example, through constraining the minimum and/or maximum number of choices that can be selected in a choiceInteraction. In some interactive delivery engines it is possible to check these constraints while the candidate is interacting with the item. As the candidate navigates around a test they may see an indication of which items have valid responses and which require attention, or the candidate may be prevented from progressing through a test until a valid response has been selected/constructed. (See validateResponses for how to enforce this mode of operation during a test.) In some cases, delivery engines may even provide interactive controls that eliminate certain types of invalid response, for example, by restricting the number of choices that can be selected to prevent it exceeding the maximum specified for the interaction.

Given the possibility that a candidate may place an interaction into a state that does not satisfy these additional constraints, item authors must ensure that the response processing rules (if provided) deal appropriately with invalid values of the response variables.

Abstract class : `inlineInteraction` (flow, inline, interaction)

Derived classes:
`endAttemptInteraction`, `inlineChoiceInteraction`, `textEntryInteraction`

An interaction that appears inline.

Abstract class : `blockInteraction` (block, flow, interaction)

Derived classes:
`associateInteraction`, `choiceInteraction`, `drawingInteraction`, `extendedTextInteraction`, `gapMatchInteraction`, `graphicInteraction`, `hottextInteraction`, `matchInteraction`, `mediaInteraction`, `orderInteraction`, `sliderInteraction`, `uploadInteraction`

An interaction that behaves like a block in the content model. Most interactions are of this type.

Contains : prompt [0..1]
An optional prompt for the interaction.

Class : `prompt` (bodyElement)

Associated classes:
`blockInteraction`

Contains : inlineStatic [*]
A prompt must not contain any nested interactions.

Abstract class : `choice` (bodyElement)

Derived classes:
`associableChoice`, `hotspotChoice`, `hottext`, `inlineChoice`, `simpleChoice`

Many of the interactions involve choosing one or more predefined choices. These choices all have the following attributes in common:

Attribute : `identifier [1]:` identifier
The identifier of the choice. This identifier must not be used by any other choice *or item variable*.

Attribute : `fixed [0..1]:` boolean = false
If fixed is true for a choice then the position of this choice within the interaction must not be changed by the delivery engine even if the immediately enclosing interaction supports the shuffling of choices. If no value is specified then the choice is free to be shuffled.

In Item Templates, the visibility of choices can be controlled by setting the value(s) of an associated template variable during template processing. For information about item templates see Item Templates.

Attribute : `templateIdentifier [0..1]:` identifier
The identifier of a *template* variable that must have a base-type of identifier and be either single of multiple cardinality. When the associated interaction is part of an Item Template the value of the identified template variable is used to control the visibility of the choice. When a choice is hidden it is not selectable and its content is not visible to the candidate unless otherwise stated.

Attribute : `showHide [0..1]:` showHide = show
The showHide attribute determines how the visibility of the choice is controlled. If set to show then the choice is hidden by default and shown only if the associated template variable matches, or contains, the identifier of the choice. If set to hide then the choice is shown by default and hidden if the associated template variable matches, or contains, the choice's identifier.

Abstract class : `associableChoice` (choice)

Derived classes:
associableHotspot, gap, gapChoice, simpleAssociableChoice

Other interactions involve associating pairs of predefined choices. These choices all have the following attribute in common:

Attribute : `matchGroup [0..*]`: identifier
A set of choices that this choice may be associated with, all others are excluded. If no matchGroup is given, or if it is empty, then all other choices may be associated with this one subject to their own matching constraints.

7.1. Simple Interactions

Class : `choiceInteraction` (blockInteraction)

The choice interaction presents a set of choices to the candidate. The candidate's task is to select one or more of the choices, up to a maximum of maxChoices. There is no corresponding minimum number of choices. The interaction is always initialized with no choices selected.

The choiceInteraction must be bound to a response variable with a baseType of identifier and single or multiple cardinality.

Attribute : `shuffle [1]`: boolean = false
If the shuffle attribute is true then the delivery engine must randomize the order in which the choices are presented subject to the fixed attribute.

Attribute : `maxChoices [1]`: integer = 1
The maximum number of choices that the candidate is allowed to select. If maxChoices is 0 then there is no restriction. If maxChoices is greater than 1 (or 0) then the interaction must be bound to a response with multiple cardinality.

Attribute : `minChoices [0..1]`: integer = 0
The minimum number of choices that the candidate is required to select to form a valid response. If minChoices is 0 then the candidate is not *required* to select any choices. minChoices must be less than or equal to the limit imposed by maxChoices.

Contains : simpleChoice [1..*]
An ordered list of the choices that are displayed to the user. The order is the order of the choices presented to the user, unless shuffle is true.

Class : `orderInteraction` (blockInteraction)

In an order interaction the candidate's task is to reorder the choices, the order in which the choices are displayed initially is significant. By default the candidate's task is to order all of the

choices but a subset of the choices can be requested using the maxChoices and minChoices attributes. When specified, the candidate must select a subset of the choices *and* impose an ordering on them.

If a default value is specified for the response variable associated with an order interaction then its value should be used to override the order of the choices specified here.

By its nature, an order interaction may be difficult to render in an unanswered state, especially in the default case where all choices are to be ordered. Implementors should be aware of the issues concerning the use of default values described in the section on Response Variables.

The orderInteraction must be bound to a response variable with a baseType of identifier and ordered cardinality only.

Contains : simpleChoice [1..*]
An ordered list of the choices that are displayed to the user. The order is the initial order of the choices presented to the user, unless shuffle is true.

Attribute : `shuffle [1]:` boolean = false
If the shuffle attribute is true then the delivery engine must randomize the order in which the choices are initially presented subject to the fixed attribute.

Attribute : `minChoices [0..1]:` integer
The minimum number of choices that the candidate must select and order to form a valid response to the interaction. If specified, minChoices must be 1 or greater but must not exceed the number of choices available. If unspecified, *all* of the choices must be ordered and maxChoices is ignored.

Attribute : `maxChoices [0..1]:` integer
The maximum number of choices that the candidate may select and order when responding to the interaction. Used in conjunction with minChoices, if specified, maxChoices must be greater than or equal to minChoices and must not exceed the number of choices available. If unspecified, *all* of the choices may be ordered.

Attribute : `orientation [0..1]:` orientation
The orientation attribute provides a hint to rendering systems that the ordering has an inherent vertical or horizontal interpretation.

Class : `simpleChoice (`choice)

Associated classes:
`orderInteraction, choiceInteraction`

Contains : flowStatic [*]

simpleChoice is a choice that contains flowStatic objects. A simpleChoice must not contain any nested interactions.

Class : `associateInteraction` (blockInteraction)

An associate interaction is a blockInteraction that presents candidates with a number of choices and allows them to create associations between them.

The associateInteraction must be bound to a response variable with base-type pair and either single or multiple cardinality.

Attribute : `shuffle [1]:` boolean = false
If the shuffle attribute is true then the delivery engine must randomize the order in which the choices are presented subject to the fixed attribute of the choice.

Attribute : `maxAssociations [1]:` integer = 1
The maximum number of associations that the candidate is allowed to make. If maxAssociations is 0 then there is no restriction. If maxAssociations is greater than 1 (or 0) then the interaction must be bound to a response with multiple cardinality.

Attribute : `minAssociations [0..1]:` integer = 0
The minimum number of associations that the candidate is required to make to form a valid response. If minAssociations is 0 then the candidate is not *required* to make any associations. minAssociations must be less than or equal to the limit imposed by maxAssociations.

Contains : simpleAssociableChoice [1..*]
An ordered set of choices.

Class : `matchInteraction` (blockInteraction)

A match interaction is a blockInteraction that presents candidates with two sets of choices and allows them to create associates between pairs of choices in the two sets, but not between pairs of choices in the same set. Further restrictions can still be placed on the allowable associations using the matchMax and matchGroup attributes of the choices.

The matchInteraction must be bound to a response variable with base-type directedPair and either single or multiple cardinality.

Attribute : `shuffle [1]:` boolean = false
If the shuffle attribute is true then the delivery engine must randomize the order in which the choices are presented within each set, subject to the fixed attribute of the choices themselves.

Attribute : `maxAssociations [1]:` integer = 1
The maximum number of associations that the candidate is allowed to make. If maxAssociations is 0 then there is no restriction. If maxAssociations is greater than 1 (or 0) then the interaction must be bound to a response with multiple cardinality.

Attribute : `minAssociations [0..1]:` integer = 0
The minimum number of associations that the candidate is required to make to form a valid response. If minAssociations is 0 then the candidate is not *required* to make any associations. minAssociations must be less than or equal to the limit imposed by maxAssociations.

Contains : simpleMatchSet [2]
The two sets of choices, the first set defines the source choices and the second set the targets.

Class : `simpleAssociableChoice (`associableChoice)

Associated classes:
`associateInteraction, simpleMatchSet`

Attribute : `matchMax [1]:` integer
The maximum number of choices this choice may be associated with. If matchMax is 0 then there is no restriction.

Attribute : `matchMin [0..1]:` integer = 0
The minimum number of choices this choice must be associated with to form a valid response. If matchMin is 0 then the candidate is not required to associate this choice with any others at all. matchMin must be less than or equal to the limit imposed by matchMax.

Contains : flowStatic [*]

associableChoice is a choice that contains flowStatic objects, it must *not* contain nested interactions.

Class : `simpleMatchSet`

Associated classes:
`matchInteraction`

Contains : simpleAssociableChoice [*]
An ordered set of choices for the set.

Class : `gapMatchInteraction (`blockInteraction)

A gap match interaction is a blockInteraction that contains a number gaps that the candidate can fill from an associated set of choices. The candidate must be able to review the content with the gaps filled in context, as indicated by their choices.

The gapMatchInteraction must be bound to a response variable with base-type directedPair and either single or multiple cardinality, depending on the number of gaps. The choices represent the source of the pairing and gaps the targets. Each gap can have at most one choice associated with it. The maximum occurrence of the choices is controlled by the matchMax attribute of gapChoice.

Attribute : `shuffle [1]:` boolean = false
If the shuffle attribute is true then the delivery engine must randomize the order in which the choices are presented (not the gaps), subject to the fixed attribute of the choices themselves.

Contains : gapChoice [1..*]
An ordered list of choices for filling the gaps. There may be fewer choices than gaps if required.

Contains : blockStatic [1..*]
The content of the interaction is simply a piece of content that contains the gaps. If the block contains more than one gap then the interaction must be bound to a response with multiple cardinality.

Class : `gap (`associableChoice, inlineStatic)

gap is an inlineStatic element that must only appear within a gapMatchInteraction.

Attribute : `required [0..1]:` boolean = false
If *true* then this gap *must* be filled by the candidate in order to form a valid response to the interaction.

Abstract class : `gapChoice (`associableChoice)

Derived classes:
`gapImg, gapText`
Associated classes:
`gapMatchInteraction`

The choices that are used to fill the gaps in a gapMatchInteraction are either simple runs of text or single image objects, both derived from gapChoice.

Attribute : `matchMax [1]:` integer
The maximum number of choices this choice may be associated with. If matchMax is 0 there is no restriction.

Attribute : `matchMin [0..1]:` integer = 0
The minimum number of gaps this choice must be associated with to form a valid response. If matchMin is 0 then the candidate is not required to associate this choice with any gaps at all. matchMin must be less than or equal to the limit imposed by matchMax.

Class : `gapText (`gapChoice)

A simple run of text to be inserted into a gap by the user, may be subject to variable value substitution with printedVariable.

Contains : textOrVariable [*]

Class : gapImg (gapChoice)

Associated classes:
graphicGapMatchInteraction

A gap image contains a single image object to be inserted into a gap by the candidate.

Attribute : objectLabel [0..1]: string
An optional label for the image object to be inserted.

Contains : object [1]

7.2. Text-based Interactions

Class : inlineChoiceInteraction (inlineInteraction)

A inline choice is an inlineInteraction that presents the user with a set of choices, each of which is a simple piece of text. The candidate's task is to select one of the choices. Unlike the choiceInteraction, the delivery engine must allow the candidate to review their choice within the context of the surrounding text.

The inlineChoiceInteraction must be bound to a response variable with a baseType of identifier and single cardinality only.

Contains : inlineChoice [1..*]
An ordered list of the choices that are displayed to the user. The order is the order of the choices presented to the user unless shuffle is true.

Attribute : shuffle [1]: boolean = false
If the shuffle attribute is true then the delivery engine must randomize the order in which the choices are presented subject to the fixed attribute.

Attribute : required [0..1]: boolean = false
If *true* then a choice *must* be selected by the candidate in order to form a valid response to the interaction.

Class : inlineChoice (choice)

Associated classes:
inlineChoiceInteraction

A simple run of text to be displayed to the user, may be subject to variable value substitution with printedVariable.

Abstract class : stringInteraction

Derived classes:
extendedTextInteraction, textEntryInteraction

String interactions can be bound to numeric response variables, instead of strings, if desired.

If detailed information about a numeric response is required then the string interaction can be bound to a response variable with record cardinality. The resulting value contains the following fields:

- stringValue: the string, as typed by the candidate.
- floatValue: the numeric value of the string typed by the candidate, as a float.
- integerValue: the numeric value of the string typed by the candidate if no fractional digits or exponent were specified, otherwise NULL. An integer.
- leftDigits: the number of digits to the left of the point. An integer.
- rightDigits: the number of digits to the right of the point. An integer.
- ndp: the number of fractional digits specified by the candidate. If no exponent was given this is the same as rightDigits. An integer.
- nsf: the number of significant digits specified by the candidate. An integer.
- exponent: the integer exponent given by the candidate or NULL if none was specified.

Attribute : `base [0..1]:` integer = 10
If the string interaction is bound to a numeric response variable then the base attribute must be used to set the number base in which to interpret the value entered by the candidate.

Attribute : `stringIdentifier [0..1]:` identifier
If the string interaction is bound to a numeric response variable then the actual string entered by the candidate can also be captured by binding the interaction to a *second* response variable (of base-type string).

Attribute : `expectedLength [0..1]:` integer
The expectedLength attribute provides a hint to the candidate as to the expected overall length of the desired response. A Delivery Engine should use the value of this attribute to set the size of the response box, where applicable. *This is not a validity constraint.*

Attribute : `patternMask [0..1]:` string
If given, the pattern mask specifies a regular expression that the candidate's response must match in order to be considered valid. The regular expression language used is defined in Appendix F of [XML_SCHEMA2]. Care is needed to ensure that the format of the required input is clear to the candidate, especially when validity checking of responses is required for progression through a test. This could be done by providing an illustrative sample response in the prompt, for example.

Attribute : `placeholderText [0..1]:` string
In visual environments, string interactions are typically represented by empty boxes into which the candidate writes or types. However, in speech based environments it is helpful to have some placeholder text that can be used to vocalize the interaction. Delivery engines should use the

value of this attribute (if provided) instead of their default placeholder text when this is required. Implementors should be aware of the issues concerning the use of default values described in the section on Response Variables.

Class : `textEntryInteraction` (inlineInteraction, stringInteraction)

A textEntry interaction is an inlineInteraction that obtains a simple piece of text from the candidate. Like inlineChoiceInteraction, the delivery engine must allow the candidate to review their choice within the context of the surrounding text.

The textEntryInteraction must be bound to a response variable with single cardinality only. The baseType must be one of string, integer, or float.

Class : `extendedTextInteraction` (blockInteraction, stringInteraction)

An extended text interaction is a blockInteraction that allows the candidate to enter an extended amount of text.

The extendedTextInteraction must be bound to a response variable with baseType of string, integer, or float. When bound to response variable with single cardinality a single string of text is required from the candidate. When bound to a response variable with multiple or ordered cardinality several *separate* text strings may be required, see maxStrings below.

Attribute : `maxStrings [0..1]:` integer
The maxStrings attribute is required when the interaction is bound to a response variable that is a container. A Delivery Engine must use the value of this attribute to control the maximum number of separate strings accepted from the candidate. When multiple strings are accepted, expectedLength applies to *each* string.

Attribute : `minStrings [0..1]:` integer = 0
The minStrings attribute specifies the minimum number separate (non-empty) strings required from the candidate to form a valid response. If minStrings is 0 then the candidate is not required to enter any strings at all. minStrings must be less than or equal to the limit imposed by maxStrings. If the interaction is *not* bound to a container then there is a special case in which minStrings may be 1. In this case the candidate must enter a non-empty string to form a valid response. More complex constraints on the form of the string can be controlled with the patternMask attribute.

Attribute : `expectedLines [0..1]:` integer
The expectedLines attribute provides a hint to the candidate as to the expected number of lines of input required. A Delivery Engine should use the value of this attribute to set the size of the response box, where applicable. *This is not a validity constraint.*

Attribute : `format [0..1]:` textFormat = plain
Used to control the format of the text entered by the candidate. See textFormat below. This attribute affects the way the value of the associated response variable should be interpreted by

response processing engines and also controls the way it should be captured in the delivery engine.

Enumeration: `textFormat`

`plain`
Indicates that the text to be entered by the candidate is plain text. This format is suitable for short unstructured responses. Delivery engines *should* preserve white-space characters in candidate input except where a response consists only of white-space characters, in which case it should be treated as an empty string (NULL).

`preFormatted`
Indicates that the text to be entered by the candidate is pre-formatted and should be rendered in a way consistent with the definition of pre in [XHTML]. Delivery engines *must* preserve white-space characters except where a response consists only of white-space characters, in which case it should be treated as an empty string (NULL).

`xhtml`
Indicates that the text to be entered by the candidate is structured text. The value of the response variable is text marked up in XHTML. The delivery engine should present an interface suitable for capturing structured text, this might be plain typed text interpreted with a set of simple text markup conventions such as those used in wiki page editors or a complete WYSIWYG editor.

Class : `hottextInteraction` (blockInteraction)

The hottext interaction presents a set of choices to the candidate represented as selectable runs of text embedded within a surrounding context, such as a simple passage of text. Like choiceInteraction, the candidate's task is to select one or more of the choices, up to a maximum of maxChoices. The interaction is initialized from the defaultValue of the associated response variable, a NULL value indicating that no choices are selected (the usual case).

The hottextInteraction must be bound to a response variable with a baseType of identifier and single or multiple cardinality.

Attribute : `maxChoices [1]:` integer = 1
The maximum number of choices that can be selected by the candidate. If matchChoices is 0 there is no restriction. If maxChoices is greater than 1 (or 0) then the interaction must be bound to a response with multiple cardinality.

Attribute : `minChoices [0..1]:` integer = 0
The minimum number of choices that the candidate is required to select to form a valid response. If minChoices is 0 then the candidate is not *required* to select any choices. minChoices must be less than or equal to the limit imposed by maxChoices.

Contains : blockStatic [1..*]
The content of the interaction is simply a piece of content, such as a simple passage of text, that contains the hottext areas.

Class : hottext (choice, flowStatic, inlineStatic)

A hottext area is used within the content of an hottextInteraction to provide the individual choices. It *must not* contain any nested interactions or other hottext areas.

When a hottext choice is hidden (by the value of an associated template variable) the content of the choice must still be presented to the candidate as if it were simply part of the surrounding material. In the case of hottext, the effect of hiding the choice is simply to make the run of text unselectable by the candidate.

Contains : inlineStatic [*]

7.3. Graphical Interactions

Abstract class : hotspot

Derived classes:
associableHotspot, hotspotChoice

Some of the graphic interactions involve images with specially defined areas or *hotspots*.

Attribute : shape [1]: shape
The shape of the hotspot.

Attribute : coords [1]: coords
The size and position of the hotspot, interpreted in conjunction with the shape.

Attribute : hotspotLabel [0..1]: string256
The alternative text for this (hot) area of the image, if specified it *must* be treated in the same way as alternative text for img. For hidden hotspots this label is ignored.

Class : hotspotChoice (choice, hotspot)

Associated classes:
hotspotInteraction, graphicOrderInteraction

Class : associableHotspot (associableChoice, hotspot)

Associated classes:
graphicAssociateInteraction, graphicGapMatchInteraction

Attribute : matchMax [1]: integer
The maximum number of choices this choice may be associated with. If matchMax is 0 there is no restriction.

Attribute : `matchMin [0..1]:` integer = 0
The minimum number of choices this choice must be associated with to form a valid response. If matchMin is 0 then the candidate is not required to associate this choice with any others at all. matchMin must be less than or equal to the limit imposed by matchMax.

Abstract class : `graphicInteraction (`blockInteraction)

Derived classes:
`graphicAssociateInteraction, graphicGapMatchInteraction, graphicOrderInteraction, hotspotInteraction, selectPointInteraction`

Contains : object [1]
Each graphical interaction has an associated image which is given as an object that must be of an *image* type, as specified by the type attribute.

Class : `hotspotInteraction (`graphicInteraction)

A hotspot interaction is a graphical interaction with a corresponding set of choices that are defined as areas of the graphic image. The candidate's task is to select one or more of the areas (hotspots). The hotspot interaction should only be used when the spacial relationship of the choices with respect to each other (as represented by the graphic image) is important to the needs of the item. Otherwise, choiceInteraction should be used instead with separate material for each option.

The delivery engine must clearly indicate the selected area(s) of the image and may also indicate the unselected areas as well. Interactions with hidden hotspots are achieved with the selectPointInteraction.

The hotspot interaction must be bound to a response variable with a baseType of identifier and single or multiple cardinality.

Attribute : `maxChoices [1]:` integer = 1
The maximum number of choices that the candidate is allowed to select. If maxChoices is 0 there is no restriction. If maxChoices is greater than 1 (or 0) then the interaction must be bound to a response with multiple cardinality.

Attribute : `minChoices [0..1]:` integer = 0
The minimum number of choices that the candidate is required to select to form a valid response. If minChoices is 0 then the candidate is not *required* to select any choices. minChoices must be less than or equal to the limit imposed by maxChoices.

Contains : hotspotChoice [1..*] {ordered}
The hotspots that define the choices that can be selected by the candidate. If the delivery system does not support pointer-based selection then the order in which the choices are given must be the order in which they are offered to the candidate for selection. For example, the 'tab order' in simple keyboard navigation. If hotspots overlap then those listed first hide overlapping hotspots that appear later. The default hotspot, if defined, must appear last.

Class : `selectPointInteraction (`graphicInteraction)

Like hotspotInteraction, a select point interaction is a graphic interaction. The candidate's task is to select one or more points. The associated response *may* have an areaMapping that scores the response on the basis of comparing it against predefined areas but the delivery engine *must not* indicate these areas of the image. Only the actual point(s) selected by the candidate shall be indicated.

The select point interaction must be bound to a response variable with a baseType of point and single or multiple cardinality.

Attribute : `maxChoices [1]:` integer = 1
This attribute is interpreted as the maximum number of points that the candidate is allowed to select. If maxChoices is 0 there is no restriction. If maxChoices is greater than 1 (or 0) then the interaction must be bound to a response with multiple cardinality.

Attribute : `minChoices [0..1]:` integer = 0
The minimum number of points that the candidate is required to select to form a valid response. If minChoices is 0 then the candidate is not *required* to select any points. minChoices must be less than or equal to the limit imposed by maxChoices.

Class : `graphicOrderInteraction (`graphicInteraction)

A graphic order interaction is a graphic interaction with a corresponding set of choices that are defined as areas of the graphic image. The candidate's task is to impose an ordering on the areas (hotspots). The order hotspot interaction should only be used when the spacial relationship of the choices with respect to each other (as represented by the graphic image) is important to the needs of the item. Otherwise, orderInteraction should be used instead with separate material for each option.

The delivery engine must clearly indicate all defined area(s) of the image.

The order hotspot interaction must be bound to a response variable with a baseType of identifier and ordered cardinality.

Contains : hotspotChoice [1..*]
The hotspots that define the choices that are to be ordered by the candidate. If the delivery system does not support pointer-based selection then the order in which the choices are given must be the order in which they are offered to the candidate for selection. For example, the 'tab order' in simple keyboard navigation.

Attribute : `minChoices [0..1]:` integer
The minimum number of choices that the candidate must select and order to form a valid response to the interaction. If specified, minChoices must be 1 or greater but must not exceed the number of choices available. If unspecified, *all* of the choices must be ordered and maxChoices is ignored.

Attribute : `maxChoices [0..1]:` integer
The maximum number of choices that the candidate may select and order when responding to the interaction. Used in conjunction with minChoices, if specified, maxChoices must be greater than or equal to minChoices and must not exceed the number of choices available. If unspecified, *all* of the choices may be ordered.

Class : `graphicAssociateInteraction (`graphicInteraction)

A graphic associate interaction is a graphic interaction with a corresponding set of choices that are defined as areas of the graphic image. The candidate's task is to associate the areas (hotspots) with each other. The graphic associate interaction should only be used when the graphical relationship of the choices with respect to each other (as represented by the graphic image) is important to the needs of the item. Otherwise, associateInteraction should be used instead with separate Material for each option.

The delivery engine must clearly indicate all defined area(s) of the image.

The associateHotspotInteraction must be bound to a response variable with base-type pair and either single or multiple cardinality.

Attribute : `maxAssociations [1]:` integer = 1
The maximum number of associations that the candidate is allowed to make. If maxAssociations is 0 there is no restriction. If maxAssociations is greater than 1 (or 0) then the interaction must be bound to a response with multiple cardinality.

Contains : associableHotspot [1..*]
The hotspots that define the choices that are to be associated by the candidate. If the delivery system does not support pointer-based selection then the order in which the choices are given must be the order in which they are offered to the candidate for selection. For example, the 'tab order' in simple keyboard navigation.

Class : `graphicGapMatchInteraction (`graphicInteraction)

A graphic gap-match interaction is a graphical interaction with a set of gaps that are defined as areas (hotspots) of the graphic image and an additional set of gap choices that are defined outside the image. The candidate must associate the gap choices with the gaps in the image and be able to review the image with the gaps filled in context, as indicated by their choices. Care should be taken when designing these interactions to ensure that the gaps in the image are a suitable size to receive the required gap choices. It must be clear to the candidate which hotspot each choice has been associated with. When associated, choices must appear wholly inside the gaps if at all possible and, where overlaps are required, should not hide each other completely. If the candidate indicates the association by positioning the choice over the gap (e.g., drag and drop) the system should 'snap' it to the nearest position that satisfies these requirements.

The graphicGapMatchInteraction must be bound to a response variable with base-type directedPair and multiple cardinality. The choices represent the source of the pairing and the

gaps in the image (the hotspots) the targets. Unlike the simple gapMatchInteraction, each gap can have several choices associated with it if desired, furthermore, the same choice may be associated with an associableHotspot multiple times, in which case the corresponding directed pair appears multiple times in the value of the response variable.

Contains : gapImg [1..*]
An ordered list of choices for filling the gaps. There may be fewer choices than gaps if required.

Contains : associableHotspot [1..*]
The hotspots that define the gaps that are to be filled by the candidate. If the delivery system does not support pointer-based selection then the order in which the gaps is given must be the order in which they are offered to the candidate for selection. For example, the 'tab order' in simple keyboard navigation. The default hotspot must not be defined.

Class : `positionObjectInteraction` (interaction)

Associated classes:
`positionObjectStage`

The position object interaction consists of a single image which must be positioned on another graphic image (the stage) by the candidate. Like selectPointInteraction, the associated response *may* have an areaMapping that scores the response on the basis of comparing it against predefined areas but the delivery engine *must not* indicate these areas of the stage. Only the actual position(s) selected by the candidate shall be indicated.

The position object interaction must be bound to a response variable with a baseType of point and single or multiple cardinality. The point records the coordinates, with respect to the stage, of the center point of the image being positioned.

Attribute : `centerPoint [0..2]`: integer
The centerPoint attribute defines the point on the image being positioned that is to be treated as the center as an offset from the top-left corner of the image in horizontal, vertical order. By default this is the center of the image's bounding rectangle.

The stage on which the image is to be positioned may be shared amongst several position object interactions and is therefore defined in a class of its own: positionObjectStage.

Attribute : `maxChoices [1]`: integer = 1
The maximum number of positions (on the stage) that the image can be placed. If matchChoices is 0 there is no limit. If maxChoices is greater than 1 (or 0) then the interaction must be bound to a response with multiple cardinality.

Attribute : `minChoices [0..1]`: integer
The minimum number of positions that the image must be placed to form a valid response to the interaction. If specified, minChoices must be 1 or greater but must not exceed the limit imposed by maxChoices.

Contains : object [1]
The image to be positioned on the stage by the candidate.

Class : `positionObjectStage` (block)

Contains : object [1]
The image to be used as a stage onto which individual positionObjectInteractions allow the candidate to place their objects.

Contains : positionObjectInteraction [1..*]

7.4. Miscellaneous Interactions

Class : `sliderInteraction` (blockInteraction)

The slider interaction presents the candidate with a control for selecting a numerical value between a lower and upper bound. It must be bound to a response variable with single cardinality with a base-type of either integer or float.

Attribute : `lowerBound [1]:` float
If the associated response variable is of type integer then the lowerBound must be rounded down to the greatest integer less than or equal to the value given.

Attribute : `upperBound [1]:` float
If the associated response variable is of type integer then the upperBound must be rounded up to the least integer greater than or equal to the value given.

Attribute : `step [0..1]:` integer
The steps that the control moves in. For example, if the lowerBound and upperBound are [0,10] and step is 2 then the response would be constrained to the set of values {0,2,4,6,8,10}. If bound to an integer response the default step is 1, otherwise the slider is assumed to operate on an approximately continuous scale.

Attribute : `stepLabel [0..1]:` boolean = false
By default, sliders are labeled only at their ends. The stepLabel attribute controls whether or not each step on the slider should also be labeled. It is unlikely that delivery engines will be able to guarantee to label steps so this attribute should be treated only as request.

Attribute : `orientation [0..1]:` orientation
The orientation attribute provides a hint to rendering systems that the slider is being used to indicate the value of a quantity with an inherent vertical or horizontal interpretation. For example, an interaction that is used to indicate the value of height might set the orientation to vertical to indicate that rendering it horizontally could spuriously increase the difficulty of the item.

Attribute : `reverse [0..1]:` boolean
The reverse attribute provides a hint to rendering systems that the slider is being used to indicate the value of a quantity for which the normal sense of the upper and lower bounds is reversed. For example, an interaction that is used to indicate a depth below sea level might specify both a vertical orientation and set reverse.

Note that a slider interaction does not have a default or initial position except where specified by a default value for the associated response variable. The currently selected value, if any, *must* be clearly indicated to the candidate.

Class : `mediaInteraction (`blockInteraction)

The media interaction allows more control over the way the candidate interacts with a time-based media object and allows the number of times the media object was experienced to be reported in the value of the associated response variable, which must be of base-type integer and single cardinality.

Attribute : `autostart [1]:` boolean
The autostart attribute determines if the media object should begin as soon as the candidate starts the attempt (true) or if the media object should be started under the control of the candidate (false).

Attribute : `minPlays [0..1]:` integer = 0
The minPlays attribute indicates that the media object should be played a minimum number of times by the candidate. The techniques required to enforce this will vary from system to system, in some systems it may not be possible at all. By default there is no minimum. Failure to play the media object the minimum number of times constitutes an invalid response.

Attribute : `maxPlays [0..1]:` integer = 0
The maxPlays attribute indicates that the media object can be played at most maxPlays times—it must not be possible for the candidate to play the media object more than maxPlay times. A value of 0 (the default) indicates that there is no limit.

Attribute : `loop [0..1]:` boolean = false
The loop attribute is used to set continuous play mode. In continuous play mode, once the media object has started to play it should play continuously (subject to maxPlays).

Contains : object [1]
The media object itself.

Class : `drawingInteraction (`blockInteraction)

The drawing interaction allows the candidate to use a common set of drawing tools to modify a given graphical image (the canvas). It must be bound to a response variable with base-type file and single cardinality. The result is a file in the same format as the original image.

Contains : object [1]
The image that acts as the canvas on which the drawing takes place is given as an object which must be of an *image* type, as specified by the type attribute.

Class : `uploadInteraction` (blockInteraction)

The upload interaction allows the candidate to upload a pre-prepared file representing their response. It must be bound to a response variable with base-type file and single cardinality.

Attribute : `type [0..1]:` mimeType
The expected mime-type of the uploaded file.

Class : `customInteraction` (block, flow, interaction)

The custom interaction provides an opportunity for extensibility of this specification to include support for interactions not currently documented.

7.5. Alternative Ways to End an Attempt

Class : `endAttemptInteraction` (inlineInteraction)

The end attempt interaction is a special type of interaction which allows item authors to provide the candidate with control over the way in which the candidate terminates an attempt. The candidate can use the interaction to terminate the attempt (triggering response processing) immediately, typically to request a hint. It must be bound to a response variable with base-type boolean and single cardinality.

If the candidate invokes response processing using an endAttemptInteraction then the associated response variable is set to true. If response processing is invoked in any other way, either through a different endAttemptInteraction or through the default method for the delivery engine, then the associated response variable is set to false. The default value of the response variable is always ignored.

Attribute : `title [1]:` string
The string that should be displayed to the candidate as a prompt for ending the attempt using this interaction. This should be short, preferably one word. A typical value would be "Hint". For example, in a graphical environment it would be presented as the label on a button that, when pressed, ends the attempt.

8. Response Processing

Response processing is the process by which the Delivery Engine assigns outcomes based on the candidate's responses. The outcomes may be used to provide feedback to the candidate. Feedback is either provided immediately following the end of the candidate's attempt or it is provided at some later time, perhaps as part of a summary report on the item session.

The end of an attempt, and therefore response processing, must only take place in direct response to a user action or in response to some expected event, such as the end of a test. An item session that enters the suspended state may have values for the response variables that have yet to be submitted for response processing.

For a Non-adaptive Item the values of the outcome variables are reset to their default values (or NULL if no default is given) before *each* invocation of response processing. However, although a Delivery Engine *may* invoke response processing multiple times for a Non-adaptive Item it **must** only report the *first* set of outcomes produced or limit the number of attempts to some predefined limit agreed outside the scope of this specification.

For an Adaptive Item the values of the outcome variables are **not** reset to their defaults. A Delivery Engine that supports Adaptive Items *must* allow the candidate to revise and submit their responses for response processing and *must* only report the *last* set of outcomes produced. Furthermore, it must present **all** applicable modal **and** integrated feedback to the candidate. Subsequent response processing may take into consideration the feedback seen by the candidate when updating the session outcomes. An adaptive item can signal to the delivery engine that the candidate has completed the interaction and no more attempts are to be allowed by setting the built-in outcome variable *completionStatus* to *completed* .

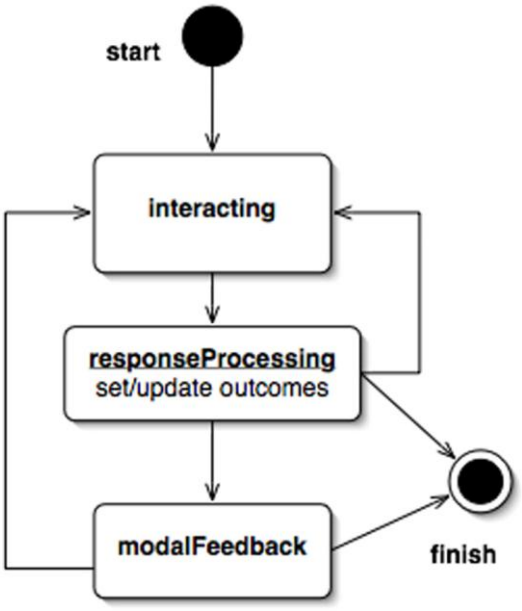

Figure 8.1 Feedback Followed by Further Interaction.

8.1. Response Processing Templates

Response processing involves the application of a set of responseRules, including the testing of responseConditions and the evaluation of expressions involving the item variables. For delivery engines that are only designed to support very simple use cases the implementation of a system for carrying out this evaluation, conditional testing and processing may pose a barrier to the adoption of the specification.

To alleviate this problem, the implementation of generalized response processing is an optional feature. Engines that don't support it can instead implement a smaller number of standard response processors called *response processing templates* described below. These templates are described using the processing language defined in this specification and are distributed (in XML form) along with it. Delivery engines that support generalized response processing do not need to implement special mechanisms to support them as a template file can be parsed directly while processing the assessmentItem that refers to it.

Delivery engines that do not support generalized response processing but do support response processing mechanisms that go beyond the standard templates described below should, where possible, define templates of their own. Authors wishing to write items for those delivery engines can then refer to these custom templates. Publishing these custom templates will then ensure that these items can be used with delivery engines that do support generalized response processing.

8.1.1. Standard Templates

Match Correct
rptemplates/match_correct.xml

Full template URI:
`http://www.imsglobal.org/question/qtiv2p1pd2/rptemplates/match_correct`

The match correct response processing template uses the match operator to match the value of a response variable *RESPONSE* with its correct value. It sets the outcome variable *SCORE* to either 0 or 1 depending on the outcome of the test. A response variable with called *RESPONSE* must have been declared and have an associated correct value. Similarly, the outcome variable *SCORE* must also have been declared. The template applies to responses of *any* baseType and cardinality though bear in mind the limitations of matching more complex data types. This template shouldn't be used for testing the numerical equality of responses with base-type float.

Note that this template always sets a value for *SCORE*, even if no *RESPONSE* was given.

Map Response
rptemplates/map_response.xml

Full template URI:
`http://www.imsglobal.org/question/qtiv2p1pd2/rptemplates/map_response`

The map response processing template uses the mapResponse operator to map the value of a response variable *RESPONSE* onto a value for the outcome *SCORE*. Note that when using the mapResponse, the SCORE needs to be of type *float*. Both variables must have been declared and *RESPONSE* must have an associated mapping. The template applies to responses of *any* baseType and cardinality. See the notes about mapResponse for details of its behavior when applied to containers.

If *RESPONSE* was NULL the *SCORE* is set to 0.0.

Map Response Point
rptemplates/map_response_point.xml

Full template URI:
`http://www.imsglobal.org/question/qtiv2p1pd2/rptemplates/map_response_point`

The map response point processing template uses the mapResponsePoint operator to map the value of a response variable *RESPONSE* onto a value for the outcome *SCORE*. Both variables must been declared and *RESPONSE* must have baseType point. See the notes about mapResponsePoint for details of its behavior when applied to containers.

If *RESPONSE* was NULL the *SCORE* is set to 0.

8.2. Generalized Response Processing

Class : `responseProcessing`

Associated classes:
`assessmentItem`

Attribute : `template [0..1]:` uri
If a template identifier is given it may be used to locate an externally defined responseProcessing template. The rules obtained from the external template may be used instead of the rules defined within the item itself, though if both are given the internal rules are still preferred.

Attribute : `templateLocation [0..1]:` uri
In practice, the template attribute may well contain a URN or the URI of a template stored on a remote web server, such as the standard response processing templates defined by this specification. When processing an assessmentItem tools working offline will not be able to obtain the template from a URN or remote URI. The templateLocation attribute provides an alternative URI, typically a relative URI to be resolved relative to the location of the assessmentItem itself, that can be used to obtain a copy of the response processing template. If a delivery system is able to determine the correct behavior from the template identifier alone the templateLocation should be ignored. For example, a delivery system may have built-in procedures for handling the standard templates defined above.

Contains : responseRule [*]

The mapping from values assigned to responses variables by the candidate onto appropriate values for the item's outcome variables is achieved through a number of rules.

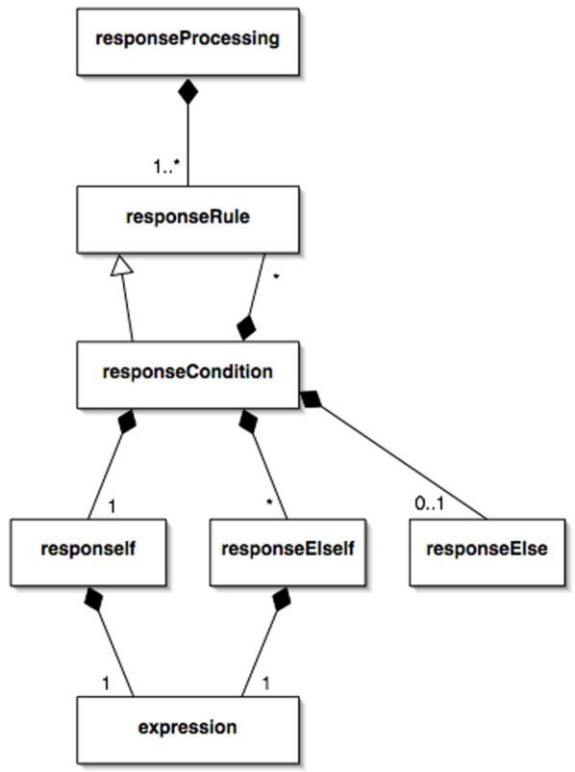

Figure 8.2 Response Processing.

Abstract class : `responseRule`

Derived classes:
`exitResponse`, `include`, `lookupOutcomeValue`, `responseCondition`, `responseProcessingFragment`, `setOutcomeValue`

Associated classes:
`responseElse`, `responseProcessingFragment`, `responseIf`, `responseProcessing`, `responseElseIf`

A response rule is either a responseCondition, a simple action or a responseProcessingFragment. Response rules define the light-weight programming language necessary for deriving outcomes from responses (i.e., scoring). Note that this programming language contains a minimal number of control structures, more complex scoring rules must be coded in other languages and referred to using a customOperator.

Class : `responseCondition` (responseRule)

Contains : responseIf [1]

Contains : responseElseIf [*]

Contains : responseElse [0..1]

If the expression given in a responseIf or responseElseIf evaluates to true then the sub-rules contained within it are followed and any following responseElseIf or responseElse parts are ignored for this response condition.

If the expression given in a responseIf or responseElseIf does not evaluate to true then consideration passes to the next responseElseIf or, if there are no more responseElseIf parts then the sub-rules of the responseElse are followed (if specified).

Class : `responseIf`

Associated classes:
`responseCondition`

Contains : expression [1]

Contains : responseRule [*]

A responseIf part consists of an expression which must have an effective baseType of boolean and single cardinality. For more information about the runtime data model employed see Expressions. It also contains a set of sub-rules. If the expression is true then the sub-rules are processed, otherwise they are skipped (including if the expression is NULL) and the following responseElseIf or responseElse parts (if any) are considered instead.

Class : `responseElseIf`

Associated classes:
`responseCondition`

Contains : expression [1]

Contains : responseRule [*]

responseElseIf is defined in an identical way to responseIf.

Class : `responseElse`

Associated classes:
`responseCondition`

Contains : responseRule [*]

Class : `setOutcomeValue (`outcomeRule, responseRule)

Attribute : `identifier [1]:` identifier
The outcome variable to be set.

Contains : expression [1]
An expression which must have an effective baseType and cardinality that matches the base-type and cardinality of the outcome variable being set.

The setOutcomeValue rule sets the value of an outcome variable to the value obtained from the associated expression. An outcome variable can be updated with reference to a previously assigned value, in other words, the outcome variable being set may appear in the expression where it takes the value previously assigned to it.

Special care is required when using the numeric base-types because floating point values can **not** be assigned to integer variables and vice-versa. The truncate, round, or integerToFloat operators must be used to achieve numeric type conversion.

Class : `lookupOutcomeValue (`outcomeRule, responseRule)

The lookupOutcomeValue rule sets the value of an outcome variable to the value obtained by looking up the value of the associated expression in the lookupTable associated with the outcome's declaration.

Attribute : `identifier [1]:` identifier
The outcome variable to be set.

Contains : expression [1]
An expression which must have single cardinality and an effective baseType of either integer, float , or duration. Integer type is required when the associated table is a matchTable.

Class : `exitResponse (`responseRule)

The exit response rule terminates response processing immediately (for this invocation).

9. Modal Feedback

Class : `modalFeedback`

Associated classes:
`assessmentItem`

Modal feedback is shown to the candidate directly following response processing. The value of an outcome variable is used in conjunction with the showHide and identifier attributes to determine whether or not the feedback is shown in a similar way to feedbackElement.

Attribute : `outcomeIdentifier [1]:` identifier

Attribute : `showHide [1]:` showHide

Attribute : `identifier [1]:` identifier

Attribute : `title [0..1]:` string
Delivery engines are not required to present the title to the candidate but may do so, for example as the title of a modal pop-up window.

Contains : flowStatic [*]
The content of the modalFeedback must not contain any interactions.

10. Item Templates

Item templates are templates that can be used for producing large numbers of similar items. Such items are often called cloned items. Item templates can be used to produce items by special purpose Cloning Engines or, where delivery engines support them, be used directly to produce a dynamically chosen clone at the start of an item session.

Each item cloned from an item template is identical except for the values given to a set of template variables. An assessmentItem is therefore an item template if it contains one or more templateDeclarations and a set of templateProcessing rules for assigning them values.

A cloning engine that creates cloned items must assign a different identifier to each clone and record the values of the template variables used to create it. A report of an item session with such a clone can then be transformed into an equivalent report for the original item template by substituting the item template's identifier for the cloned item's identifier and adding the values of the template variables to the report.

Class : `templateDeclaration` (variableDeclaration)

Associated classes:
`assessmentItem`

Template declarations declare item variables that are to be used specifically for the purposes of cloning items. They can have their value set only during templateProcessing. They are referred to within the itemBody in order to individualize the clone and possibly also within the responseProcessing rules if the cloning process affects the way the item is scored.

Template variables are instantiated as part of an item session. Their values are initialized during templateProcessing and thereafter behave as constants within the session.

Attribute : `paramVariable [0..1]:` boolean = false
This attribute determines whether or not the template variable's value should be substituted for object parameter values that match its name. See param for more information.

Attribute : `mathVariable [0..1]:` boolean = false
This attribute determines whether or not the template variable's value should be substituted for identifiers that match its name in MathML expressions. See Combining Template Variables and MathML for more information.

10.1. Using Template Variables in the Item's Body

Template variables can be referred to by printedVariable objects in the item body. The value of the template variable is used to create an appropriate run of text that is displayed. Template variables can also be used to conditionally control content through the two templateElements in a similar way to outcome variables with feedbackElements. Finally, template variables can be used to control the visibility of choices in interactions.

Abstract class : `templateElement` (bodyElement)

Derived classes:
templateBlock, templateInline

Attribute : `templateIdentifier [1]:` identifier
The identifier of a template variable that must have a base-type of identifier and be of either single or multiple cardinality. The visibility of the templateElement is controlled by the value of the variable.

Attribute : `showHide [1]:` showHide = show

Attribute : `identifier [1]:` identifier

The showHide and identifier attributes determine how the visibility of the templateElement is controlled in the same way as the similarly named showHide and identifier attributes of feedbackElement.

A template element must not contain any interactions, either directly or indirectly.

Class : `templateBlock` (blockStatic, flowStatic, templateElement)

Contains : blockStatic [*]

Class : `templateInline` (flowStatic, inlineStatic, templateElement)

Contains : inlineStatic [*]

10.2. Using Template Variables in Operator Attributes Values

Some of the operators used in expressions have attributes to control their behavior. The value of a template variable evaluated at the time the operator itself is evaluated, can be used as the value of some such attributes by using a reference to a template variable of the appropriate baseType and cardinality instead of a value from the value space of the attribute's target type. Attributes that support this type of attribute value substitution are declared with one of the following types.

When binding variable references as strings the variable's identifier must be enclosed in braces. For example, the string "{myVariable}" refers to the template variable with identifier *myVariable*. A references *must* match the identifier in a corresponding templateDeclaration.

Datatype: `integerOrTemplateRef`

The type used for attributes of integer type that may be replaced with the value of a template variable of base type integer and single cardinality.

Datatype: `floatOrTemplateRef`

The type used for attributes of float type that may be replaced with the value of a template variable of base type float and single cardinality.

Datatype: `stringOrTemplateRef`

The type used for attributes of string type that may be replaced with the value of a template variable of base type string and single cardinality.

When template references are bound to strings (using the mandatory binding described above) there is a potential ambiguity. Therefore, if a *string* attribute appears to be a reference to a template variable but there is no variable with the given name it should be treated simply as string value.

10.3. Template Processing

Class : `templateProcessing`

Associated classes:
`assessmentItem`

Contains : templateRule [1..*]
Template processing consists of one or more templateRules that are followed by the cloning engine or delivery system in order to assign values to the template variables. Template processing is identical in form to responseProcessing except that the purpose is to assign values to template variables, not outcome variables.

Abstract class : `templateRule`

Derived classes:
`exitTemplate`, `setCorrectResponse`, `setDefaultValue`, `setTemplateValue`, `templateCondition`
Associated classes:
`templateProcessing`, `templateElseIf`, `templateIf`, `templateElse`

A template rule is either a templateCondition or a simple action. Template rules define the light-weight programming language necessary for creating cloned items. Note that this programming language contains a minimal number of control structures, more complex cloning rules are outside the scope of this specification.

An expression used in a templateRule must not refer to the value of a response variable or outcome variable. It may only refer to the values of the template variables.

Class : `templateCondition` (templateRule)

Contains : templateIf [1]

Contains : templateElseIf [*]

Contains : templateElse [0..1]

If the expression given in the templateIf or templateElseIf evaluates to true then the sub-rules contained within it are followed and any following templateElseIf or templateElse parts are ignored for this template condition.

If the expression given in the templateIf or templateElseIf does not evaluate to true then consideration passes to the next templateElseIf or, if there are no more templateElseIf parts then the sub-rules of the templateElse are followed (if specified).

Class : `templateIf`

Associated classes:
`templateCondition`

Contains : expression [1]

Contains : templateRule [*]

A templateIf part consists of an expression which must have an effective baseType of boolean and single cardinality. For more information about the runtime data model employed see Expressions. It also contains a set of sub-rules. If the expression is true then the sub-rules are processed, otherwise they are skipped (including if the expression is NULL) and the following templateElseIf or templateElse parts (if any) are considered instead.

Class : `templateElseIf`

Associated classes:
`templateCondition`

Contains : expression [1]

Contains : templateRule [*]

templateElseIf is defined in an identical way to templateIf.

Class : `templateElse`

Associated classes:
`templateCondition`

Contains : templateRule [*]

Class : `setTemplateValue (`templateRule)

Attribute : `identifier [1]:` identifier
The template variable to be set.

Contains : expression [1]
An expression which must have an effective baseType and cardinality that matches the base-type and cardinality of the template variable being set.

The setTemplateValue rule sets the value of a template variable to the value obtained from the associated expression. A template variable can be updated with reference to a previously assigned value, in other words, the template variable being set may appear in the expression where it takes the value previously assigned to it.

Class : `setCorrectResponse` (templateRule)

Attribute : `identifier [1]:` identifier
The response variable to have its correct value set.

Contains : expression [1]

Class : `setDefaultValue` (templateRule)

Attribute : `identifier [1]:` identifier
The response variable or outcome variable to have its default value set.

Contains : expression [1]

Class : `exitTemplate` (templateRule)

The exit template rule terminates template processing immediately.

11. Tests

A *test* is represented by the assessmentTest class.

Class : `assessmentTest`

A test is a group of assessmentItems with an associated set of rules that determine which of the items the candidate sees, in what order, and in what way the candidate interacts with them. The rules describe the valid paths through the test, *when* responses are submitted for response processing and when (if at all) feedback is to be given.

Attribute : `identifier [1]:` string
The principle identifier of the test. This identifier must have a corresponding entry in the test's meta-data. See Meta-data and Usage Data for more information.

Attribute : `title [1]:` string
The title of an assessmentTest is intended to enable the test to be selected outside of any test session. Therefore, delivery engines may reveal the title to candidates at any time, but are not required to do so.

Attribute : `toolName [0..1]:` string256
The tool name attribute allows the tool creating the test to identify itself. Other processing systems may use this information to interpret the content of application specific data, such as labels on the elements of the test rubric.

Attribute : `toolVersion [0..1]:` string256
The tool version attribute allows the tool creating the test to identify its version. This value must only be interpreted in the context of the toolName.

Contains : outcomeDeclaration [*]
Each test has an associated set of outcomes. The values of these outcomes are set by the test's outcomeProcessing rules.

Contains : timeLimits [0..1]
Optionally controls the amount of time a candidate is allowed for the entire test.

Contains : testPart [1..*]
Each test is divided into one or more parts which may in turn be divided into sections, sub-sections, and so on. A testPart represents a major division of the test and is used to control the basic mode parameters that apply to all sections and sub-sections within that part.

Contains : outcomeProcessing [0..1]
The set of rules used for calculating the values of the test outcomes.

Contains : testFeedback [*]
Contains the test-level feedback controlled by the test outcomes.

11.1. Navigation and Submission

This specification defines two ways in which the overall behavior of each test part can be controlled: the navigation mode and the submission mode.

Enumeration: `navigationMode`

```
linear

nonlinear
```

The navigation mode determines the general paths that the candidate may take. A testPart in *linear* mode restricts the candidate to attempt each item *in turn*. Once the candidate moves on they are **not** permitted to return. A testPart in *nonlinear* mode removes this restriction - the candidate is free to navigate to any item in the test at any time. Test delivery systems are free to implement their own user interface elements to facilitate navigation provided they honor the navigation mode currently in effect. A test delivery system may implement nonlinear mode simply by providing a method to step forward or backwards through the test part.

Enumeration: `submissionMode`

```
individual

simultaneous
```

The submission mode determines when the candidate's responses are submitted for response processing. A testPart in *individual* mode requires the candidate to submit their responses on an item-by-item basis. In *simultaneous* mode the candidate's responses are all submitted together at the end of the testPart.

The choice of submission mode determines the states through which each item's session can pass during the test. In simultaneous mode, response processing cannot take place until the testPart is complete so each item session passes between the *interacting* and *suspended* states only. By definition the candidate can take one and only one attempt at each item and feedback cannot be seen during the test. Whether or not candidates can return to review their responses and/or any item-level feedback *after* the test, is outside the scope of this specification. Simultaneous mode is typical of paper-based tests.

In individual mode, response processing may take place during the test and the item session may pass through any of the states described in Items, subject to the itemSessionControl settings in force. Care should be taken when designing user interfaces for systems that support *nonlinear* navigation mode in combination with *individual* submission. With this combination candidates may change their responses for an item and then leave it in the *suspended* state by navigating to a different item in the same part of the test. Test delivery systems need to make it clear to candidates that there are unsubmitted responses (akin to unsaved changes in a traditional document editing system) at the end of the test part. A test delivery system may force candidates

to submit or discard such responses *before* moving to a different item in *individual* mode if this is more appropriate.

Class : `testPart`

Associated classes:
`assessmentTest`

Attribute : `identifier [1]:` identifier
The identifier of the test part must be unique within the test and must not be the identifier of any assessmentSection or assessmentItemRef.

Attribute : `navigationMode [1]:` navigationMode

Attribute : `submissionMode [1]:` submissionMode

Contains : preCondition [*]
An optional set of conditions evaluated during the test, that determine if this part is to be skipped.

Contains : branchRule [*]
An optional set of rules, evaluated during the test, for setting an alternative target as the next part of the test.

Contains : itemSessionControl [0..1]
Parameters used to control the allowable states of each item session in this part. These values may be overridden at section and item level.

Contains : timeLimits [0..1]
Optionally controls the amount of time a candidate is allowed for this part of the test.

Contains : assessmentSection [1..*]
The items contained in each testPart are arranged into sections and sub-sections.

Contains : testFeedback [*]
Test-level feedback specific to this part of the test.

11.2. Test Structure

For each test session, items and sub-sections are selected and arranged into order according to rules defined in the containing section. This process of selection and ordering defines a basic structure for each part of the test on a per-session basis. The paths that a candidate may take through this structure are then controlled by the mode settings for the test part and possibly by further preConditions or branchRules evaluated during the test session itself.

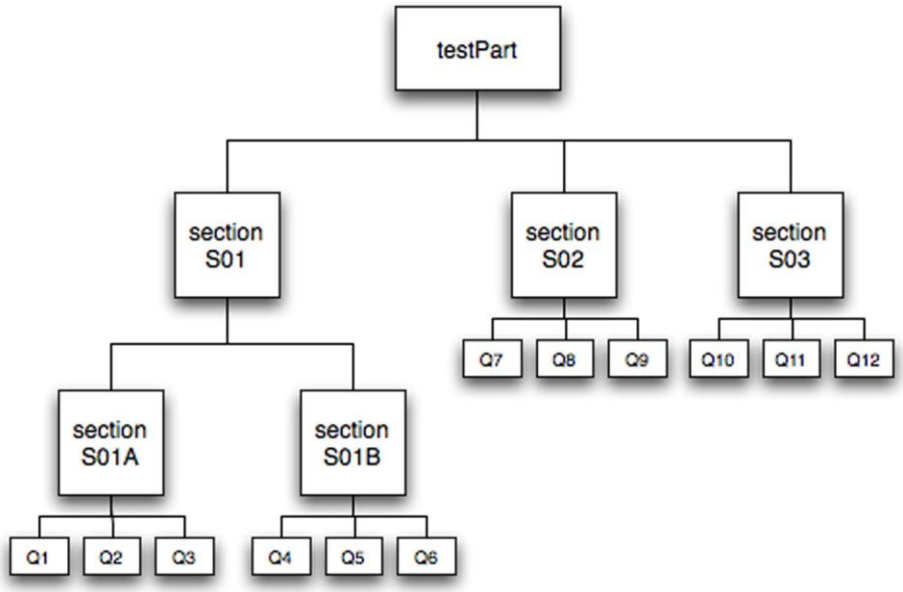

Figure 11.1 Selection and Ordering.

The figure illustrates part of a test and the way the items are structured into sections and sub-sections.

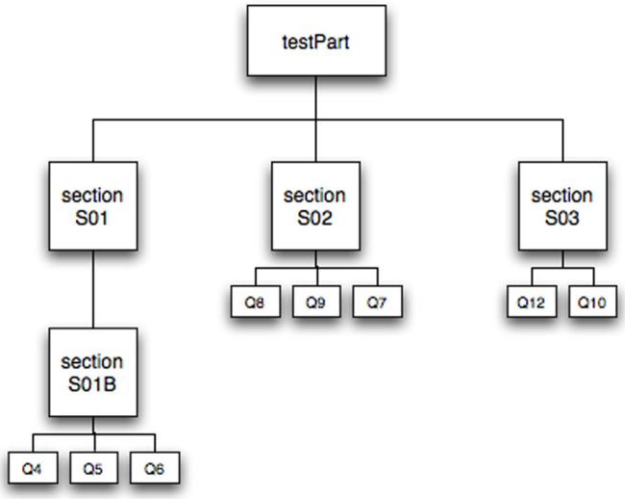

Figure 11.2 Selection and Ordering.

The second figure illustrates a specific instance of the same test part after the application of selection and ordering rules. A rule in section S01 selects just one of S01A and S01B, a rule in S02 shuffles the order of the items contained by it and, finally, rules in S03 select 2 out of the 3 items it contains *and* shuffles the result.

Class : `selection`

Associated classes:
`assessmentSection`

The *selection* class specifies the rules used to select the child elements of a section for each test session. If no selection rules are given we assume that all elements are to be selected.

Attribute : `select [1]:` integer
The number of child elements to be selected. Sub-sections always count as 1, regardless of how many child elements they have and whether or not they are visible. The number of children to select may exceed the number of child elements defined only if withReplacement is *true*.

Attribute : `withReplacement [0..1]:` boolean = false
When selecting child elements each element is normally eligible for selection once only. In other words, when selecting 3 elements from {A,B,C,D} the possible outcomes are {A,B,C}, {A,B,D}, {A,C,D}, and {B,C,D}. By setting withReplacement to *true* each element becomes eligible for selection multiple times. Selecting 3 nodes from {A,B,C,D} can then result in combinations such as {A,A,A}, {A,A,B} and so on.

The selection class also provides an opportunity for extensions to this specification to include support for more complex selection algorithms.

Class : `ordering`

Associated classes:
`assessmentSection`

The *ordering* class specifies the rule used to arrange the child elements of a section following selection. If no ordering rule is given we assume that the elements are to be ordered in the order in which they are defined.

Attribute : `shuffle [1]:` boolean = false
If *true* causes the order of the child elements to be randomized, if *false* uses the order in which the child elements are defined.

A sub-section is always treated as a single block for selection but the way it is treated when shuffling depends on its visibility. A visible sub-section is always treated as a single block but an invisible sub-section is only treated as a single block if its keepTogether attribute is *true*. Otherwise, the child elements of the invisible sub-section are mixed into the parent's selection prior to shuffling.

The ordering class also provides an opportunity for extensions to this specification to include support for more complex ordering algorithms.

The selection and ordering rules define a *sequence* of items for each *instance* of the test. The sequence starts with the first item of the first section of the first test part and continues through to the last item of the last section of the last test part. This sequence is constant throughout the test. Normally this is the logical sequence perceived by the candidate but the use of preConditions and/or branchRules can affect the specific path taken.

The use of selection with replacement enables two or more instances of an item referred to by the *same* assessmentItemRef to appear in the sequence of items for a test. It is therefore an error to make such an item the target of a branchRule. Furthermore, when reporting test results the sequence number of each item must also be reported to avoid ambiguity. See Results Reporting.

Abstract class : `sectionPart`

Derived classes:
`assessmentItemRef`, `assessmentSection`, `include`
Associated classes:
`assessmentSection`

Sections group together individual item references and/or sub-sections. A number of common parameters are shared by both types of child element.

Attribute : `identifier [1]:` identifier
The identifier of the section or item reference must be unique within the test and must not be the identifier of any testPart.

Attribute : `required [0..1]:` boolean = false
If a child element is *required* it must appear (at least once) in the selection. It is in error if a section contains a selection rule that selects fewer child elements than the number of required elements it contains.

Attribute : `fixed [0..1]:` boolean = false
If a child element is *fixed* it must never be shuffled. When used in combination with a selection rule fixed elements do not have their position fixed until after selection has taken place. For example, selecting 3 elements from {A,B,C,D} without replacement might result in the selection {A,B,C}. If the section is subject to shuffling but B is fixed then permutations such as {A,C,B} are not allowed whereas permutations like {C,B,A} are.

Contains : preCondition [*]
An optional set of conditions evaluated during the test, that determine if the item or section is to be skipped (in *nonlinear* mode, pre-conditions are ignored).

Contains : branchRule [*]
An optional set of rules, evaluated during the test, for setting an alternative target as the next item or section (in *nonlinear* mode, branch rules are ignored).

Contains : itemSessionControl [0..1]
Parameters used to control the allowable states of each item session (may be overridden at sub-section or item level).

Contains : timeLimits [0..1]
Optionally controls the amount of time a candidate is allowed for this item or section.

Class : `assessmentSection` (sectionPart)

Associated classes:
`testPart`

Attribute : `title [1]`: string
The title of the section is intended to enable the section to be selected in situations where the contents of the section are not available, for example when a candidate is browsing a test. Therefore, delivery engines may reveal the title to candidates at any time during the test but are not required to do so.

Attribute : `visible [1]`: boolean
A visible section is one that is identifiable by the candidate. For example, delivery engines might provide a hierarchical view of the test to aid navigation. In such a view, a visible section would be a visible node in the hierarchy. Conversely, an invisible section is one that is not visible to the candidate—the child elements of an invisible section appear to the candidate as if they were part of the parent section (or testPart). The visibility of a section does **not** affect the visibility of its child elements. The visibility of each section is determined solely by the value of its own visible attribute.

Attribute : `keepTogether [0..1]`: boolean = true
An invisible section with a parent that is subject to shuffling can specify whether or not its children, which will appear to the candidate as if they were part of the parent, are shuffled as a block or mixed up with the other children of the parent section.

Contains : selection [0..1]
The rules used to select which children of the section are to be used for each instance of the test.

Contains : ordering [0..1]
The rules used to determine the order in which the children of the section are to be arranged for each instance of the test.

Each child section has its own selection and ordering rules followed before those of its parent. A child section may shuffle the order of its own children while still requiring that they are kept together when shuffling the parent section.

Contains : rubricBlock [*]
Section rubric is presented to the candidate with each item contained (directly or indirectly) by

the section. As sections are nestable the rubric presented for each item is the concatenation of the rubric blocks from the top-most section down to the item's immediately enclosing section.

Contains : sectionPart [*]
An empty section is assumed to be one that uses a selection extension mechanism to describe its contents.

Class : `assessmentItemRef (`sectionPart)

Items are incorporated into the test by *reference* and not by direct aggregation. Note that the identifier of the reference need not have any meaning outside the test. In particular it is not required to be unique in the context of any catalog, or be represented in the item's meta-data. The syntax of this identifier is more restrictive than that of the identifier attribute of the assessmentItem itself.

Attribute : `href [1]:` uri
The uri used to refer to the item's file (e.g., elsewhere in the same content package). There is no requirement that this be unique. A test may refer to the same item multiple times within a test. Note however that each reference *must* have a unique identifier.

Attribute : `category [*]:` identifier
Items can optionally be assigned to one or more categories. Categories are used to allow custom sets of item outcomes to be aggregated during outcomes processing.

Contains : variableMapping [*]
Variable mappings are used to alter the name of an item outcome for the purposes of this test.

Contains : weight [*]
Weights allow custom values to be defined for scaling an item's outcomes.

Contains : templateDefault [*]
A template default is used to alter the default value of a template variable declared by the item based on an expression evaluated at test-level. See templateDefault for more information.

Class : `weight`

Associated classes:
`assessmentItemRef`

The contribution of an individual item score to an overall test score typically varies from test to test. The score of the item is said to be *weighted*. Weights are defined as part of each reference to an item (assessmentItemRef) within a test.

Attribute : `identifier [1]:` identifier
An item can have any number of weights, each one is given an identifier that is used to refer to the weight in outcomes processing. (See the variable and testVariables definitions.)

Attribute : `value [1]:` float
Weights are floating point values. Weights can be applied to outcome variables of base type float or integer. The weight is applied at the time the variable's value is evaluated during outcomes processing. The result is always treated as having base type float even if the variable itself was declared as being of base type integer.

Class : `variableMapping`

Associated classes:
`assessmentItemRef`

Variable mappings allow outcome variables declared with the name sourceIdentifier in the corresponding item to be treated as if they were declared with the name targetIdentifier during outcomeProcessing. Use of variable mappings allows more control over the way outcomes are aggregated when using testVariables.

Attribute : `sourceIdentifier [1]:` identifier

Attribute : `targetIdentifier [1]:` identifier

Class : `templateDefault`

Associated classes:
`assessmentItemRef`

The default value of a template variable in an item can be overridden based on the test context in which the template is instantiated. The value is obtained by evaluating an expression defined within the *reference* to the item at test level and which may therefore depend on the values of variables taken from other items in the test or from outcomes defined at test level itself.

For consistent results it is vital that the expression is evaluated at the correct time. When the assessmentItemRef occurs in a testPart navigated in linear mode the expression is evaluated immediately prior to the start of the first attempt, *after* any pre-conditions are evaluated and acted upon but before the templateProcessing rules of the item itself are followed. In nonlinear mode the expression is evaluated at the start of the testPart. In both cases, the *timing* is unaffected by the submissionMode in effect. Care needs to be taken to ensure that values of response variables are not used before they have been submitted and that outcome variables are not used before their values have been set by the corresponding response or outcomes processing steps.

Attribute : `templateIdentifier [1]:` identifier
The identifier of the template variable affected.

Contains : expression [1]
An expression which must result in a value with baseType and cardinality matching the declaration of the associated variable's templateDeclaration.

The facility of overriding template defaults allows item templates to be linked to other items (or templates) within a test. A candidate response from one item can be used directly to affect the presentation or even the behavior of an item presented after it. However once the template is instantiated it operates independently—there is no dynamic link created between the items and no concept of a shared variable space between them.

11.3. Time Limits

Class : `timeLimits`

Associated classes:
`assessmentTest`, `testPart`, `sectionPart`

Attribute : `minTime [0..1]:` duration

Attribute : `maxTime [0..1]:` duration

In the context of a specific assessmentTest an item, or group of items, may be subject to a time constraint. This specification supports both minimum and maximum time constraints. The controlled time for a single item is simply the duration of the item session as defined by the built-in response variable *duration*. For assessmentSections, testParts and whole assessmentTests the time limits relate to the durations of *all* the item sessions *plus* any other time spent navigating that part of the test. In other words, the time includes time spent in states where no item is being interacted with, such as dedicated navigation screens.

Minimum times are applicable to assessmentSections and assessmentItems only when linear navigation mode is in effect.

Delivery Engines are required to track and report the time spent on each test part when time limits are in force. If no time limit is in force for a given test part or section then the time spent may be tracked and reported but it is not required. Similarly, if no time limit is in force for an item *that is not a Time Dependent Item* then the time spent may be reported but is not required either.

The time spent on the test is recorded as if it were a built-in response variable called *duration* declared at the test-level and of base-type duration and single cardinality. Similarly, time spent on test parts or sections are treated as built-in response variables declared within each respective scope. The values of these durations can be referred to during outcomeProcessing by using the variable name *duration* prefixed with the identifier of the part or section followed by the period character. See the variable expression for further information.

12. Outcome Processing

Class : `outcomeProcessing`

Associated classes:
`assessmentTest`

Outcome processing takes place each time the candidate submits the responses for an item (when in individual submission mode) or a group of items (when in simultaneous submission mode). It happens *after* any (item level) response processing triggered by the submission. The values of the test's outcome variables are always reset to their defaults prior to carrying out the instructions described by the outcomeRules. Because outcome processing happens *each time* the candidate submits responses the resulting values of the test-level outcomes may be used to activate test-level feedback during the test or to control the behavior of subsequent parts through the use of preConditions and branchRules.

The structure of outcome processing is similar to that of responseProcessing.

Contains : outcomeRule [*]

Abstract class : `outcomeRule`

Derived classes:
`exitTest`, `include`, `lookupOutcomeValue`, `outcomeCondition`, `outcomeProcessingFragment`, `setOutcomeValue`
Associated classes:
`outcomeIf`, `outcomeElse`, `outcomeProcessingFragment`, `outcomeProcessing`, `outcomeElseIf`

Class : `outcomeCondition` (outcomeRule)

Contains : outcomeIf [1]

Contains : outcomeElseIf [*]

Contains : outcomeElse [0..1]

If the expression given in the outcomeIf or outcomeElseIf evaluates to true then the sub-rules contained within it are followed and any following outcomeElseIf or outcomeElse parts are ignored for this outcome condition.

If the expression given in the outcomeIf or outcomeElseIf does not evaluate to true then consideration passes to the next outcomeElseIf or, if there are no more outcomeElseIf parts then the sub-rules of the outcomeElse are followed (if specified).

Class : `outcomeIf`

Associated classes:
outcomeCondition

Contains : expression [1]

Contains : outcomeRule [*]

A outcomeIf part consists of an expression which must have an effective baseType of boolean and single cardinality. For more information about the runtime data model employed see Expressions. It also contains a set of sub-rules. If the expression is true then the sub-rules are processed, otherwise they are skipped (including if the expression is NULL) and the following outcomeElseIf or outcomeElse parts (if any) are considered instead.

Class : outcomeElseIf

Associated classes:
outcomeCondition

Contains : expression [1]

Contains : outcomeRule [*]

outcomeElseIf is defined in an identical way to outcomeIf.

Class : outcomeElse

Associated classes:
outcomeCondition

Contains : outcomeRule [*]

Class : exitTest (outcomeRule)

13. Test-level Feedback

Class : `testFeedback`

Associated classes:
`assessmentTest`, `testPart`

Attribute : `access [1]:` testFeedbackAccess
Test feedback is shown to the candidate either directly following outcome processing (during the test) or at the end of the testPart or assessmentTest as appropriate (referred to as atEnd).

The value of an outcome variable is used in conjunction with the showHide and identifier attributes to determine whether or not the feedback is actually shown in a similar way to feedbackElement.

Attribute : `outcomeIdentifier [1]:` identifier

Attribute : `showHide [1]:` showHide

Attribute : `identifier [1]:` identifier

Attribute : `title [0..1]:` string
Delivery engines are not required to present the title to the candidate but may do so, for example as the title of a modal pop-up window or sub-heading in a combined report.

Contains : flowStatic [*]
The content of the testFeedback must not contain any interactions.

Enumeration: `testFeedbackAccess`

```
atEnd

during
```

14. Pre-conditions and Branching

Pre-conditions and branch rules are advanced concepts used to introduce an element of adaptivity into the specification of an assessmentTest. The concepts were introduced in version 1.2 (as pre-conditions and post-conditions) but their specification was postponed until version 2.1. A primary consideration in introducing these features has been to limit the complexity of supporting delivery engines while still enabling some items (or whole sections and test parts) to be skipped depending on the candidate's responses to items presented earlier in the test.

Pre-conditions and branch rules are only applicable to parts of the test that are navigated in linear mode. In nonlinear mode they are ignored. The overarching test parts are effectively navigated in nonlinear mode and so can therefore have associated pre-conditions and branch rules.

Class : `preCondition`

Associated classes:
`testPart`, `sectionPart`

A preCondition is a simple expression attached to an assessmentSection or assessmentItemRef that must evaluate to true if the item is to be presented. Pre-conditions are evaluated at the time the associated item, section, or testPart is to be attempted by the candidate, **during the test**. They differ from rules for selection and ordering (see Test Structure) which are followed at or before the start of the test.

If the expression evaluates to false, or has a NULL value, the associated item or section is skipped.

Contains : expression [1]

Class : `branchRule`

Associated classes:
`testPart`, `sectionPart`

A branch-rule is a simple expression attached to an assessmentItemRef, assessmentSection or testPart that is evaluated **after** the item, section, or part has been presented to the candidate. If the expression evaluates to true the test jumps forward to the item, section, or part referred to by the target identifier. In the case of an item or section, the target must refer to an item or section in the same testPart that has not yet been presented. For testParts, the target must refer to another testPart.

[Comment] The above definition restricts the navigation paths through a linear test part to being trees. In other words, cycles are not allowed. In most cases, repetition can be achieved by using a section that selects withReplacement up to a suitable upper bound of repetition in combination with a preCondition or branchRule that terminates the section early when (or if) a certain

outcome has been achieved. (This technique might be used in conjunction with one or more Item Templates to achieve *drill and practice*, for example.) However, unbounded repetition is not supported. Comments are sought on whether this approach is too restrictive.

Attribute : `target [1]:` identifier
The following values are reserved and have special meaning when used as a target identifier: *EXIT_SECTION* jumps over all the remaining children of the current section to the item (or section) immediately following it; *EXIT_TESTPART* finishes the current testPart immediately and *EXIT_TEST* finishes the entire assessmentTest immediately.

Contains : expression [1]

15. Expressions

Abstract class : `expression`

Derived classes:

and, anyN, baseValue, containerSize, contains, correct, customOperator, default, delete, divide, durationGTE, durationLT, equal, equalRounded, fieldValue, gt, gte, index, inside, integerDivide, integerModulus, integerToFloat, isNull, lt, lte, mapResponse, mapResponsePoint, match, member, multiple, not, null, numberCorrect, numberIncorrect, numberPresented, numberResponded, numberSelected, or, ordered, outcomeMaximum, outcomeMinimum, patternMatch, power, product, random, randomFloat, randomInteger, round, stringMatch, substring, subtract, sum, testVariables, truncate, variable

Associated classes:

and, gt, ordered, divide, setCorrectResponse, random, responseIf, substring, member, equalRounded, outcomeIf, integerToFloat, lookupOutcomeValue, setTemplateValue, integerDivide, gte, index, durationLT, contains, durationGTE, branchRule, lt, match, patternMatch, product, multiple, power, outcomeElseIf, setDefaultValue, customOperator, stringMatch, setOutcomeValue, templateDefault, not, templateElseIf, integerModulus, subtract, responseElseIf, anyN, preCondition, round, containerSize, inside, equal, or, isNull, templateIf, lte, sum, truncate, fieldValue, delete

Expressions are used to assign values to item variables and to control conditional actions in response and template processing.

An expression can be a simple reference to the value of an itemVariable, a constant value from one of the value sets defined by baseTypes or a hierarchical expression operator. Like itemVariables, each expression can also have the special value NULL.

15.1. Built-in General Expressions

This section describes a number of built-in general expressions for handling constants, random values and the values of itemVariables. and

Class : `baseValue (`expression`)`

Attribute : `baseType [1]:` baseType
The base-type of the value.

The simplest expression returns a single value from the set defined by the given baseType.

Class : `variable (`expression`)`

Attribute : `identifier [1]:` identifier

This expression looks up the value of an itemVariable that has been declared in a corresponding variableDeclaration or is one of the built-in variables. The result has the base-type and cardinality declared for the variable subject to the type promotion of weighted outcomes (see below).

During outcomes processing, values taken from an individual item session can be looked up by prefixing the name of the item variable with the identifier assigned to the item in the assessmentItemRef, separated by a period character. For example, to obtain the value of the *SCORE* variable in the item referred to as *Q01* you would use a variable instance with identifier *Q01.SCORE*.

When looking up the value of a response variable it always takes the value assigned to it by the candidate's last submission. Unsubmitted responses are not available during expression evaluation.

The value of an item variable taken from an item instantiated multiple times *from the same assessmentItemRef* (through the use of selection withReplacement) is taken from the last instance submitted if submission is simultaneous, otherwise it is undefined.

Attribute : `weightIdentifier [0..1]:` identifier
An optional weighting to be applied to the value of the variable. Weights are defined only in the test context (and hence only in outcomes processing) and only when the item identifier prefixing technique (see above) is being used to look up the value of an *item* variable. The weight identifier refers to a weight definition in the corresponding assessmentItemRef. If no matching definition is found the weight is assumed to be 1.0.

Weights only apply to item variables with base types integer and float. If the item variable is of any other type the weight is *ignored*. All weights are treated as having base type float and the resulting value is obtained by multiplying the variable's value by the associated weight. When applying a weight to the value of a variable with base type integer the value is subject to type promotion and the result of the expression has base type **float**.

Class : `default (`expression)

Attribute : `identifier [1]:` identifier

This expression looks up the declaration of an itemVariable and returns the associated defaultValue or NULL if no default value was declared. When used in outcomes processing item identifier prefixing (see variable) may be used to obtain the default value from an individual item.

Class : `correct (`expression)

Attribute : `identifier [1]:` identifier

This expression looks up the declaration of a response variable and returns the associated correctResponse or NULL if no correct value was declared. When used in outcomes processing item identifier prefixing (see variable) may be used to obtain the correct response from an individual item.

Class : `mapResponse (`expression)

Attribute : `identifier [1]:` identifier

This expression looks up the value of a response variable and then transforms it using the associated mapping, which must have been declared. The result is a single float. If the response variable has single cardinality then the value returned is simply the mapped target value from the map. If the response variable has multiple or ordered cardinality then the value returned is *the sum* of the mapped target values. This expression cannot be applied to variables of record cardinality.

For example, if a mapping associates the identifiers {A,B,C,D} with the values {0,1,0.5,0} respectively then mapResponse will map the single value 'C' to the numeric value 0.5 and the set of values {C,B} to the value 1.5.

If a container contains multiple instances of the **same** value then that value is counted **once only**. To continue the example above {B,B,C} would still map to 1.5 and **not** 2.5.

Class : `mapResponsePoint (`expression)

Attribute : `identifier [1]:` identifier

This expression looks up the value of a response variable that must be of base-type point, and transforms it using the associated areaMapping. The transformation is similar to mapResponse except that the points are tested against each area in turn. When mapping containers each **area** can be mapped **once only**. For example, if the candidate identified two points that both fall in the same area then the mappedValue is still added to the calculated total just once.

Class : `null (`expression)

null is a simple expression that returns the NULL value—the null value is treated as if it is of any desired baseType.

Class : `randomInteger (`expression)

Selects a random integer from the specified range [min,max] satisfying min + step * n for some integer *n*. For example, with min=2, max=11 and step=3 the values {2,5,8,11} are possible.

Attribute : `min [1]:` integerOrTemplateRef = 0

Attribute : `max [1]:` integerOrTemplateRef

Attribute : `step [0..1]:` integerOrTemplateRef = 1

Class : `randomFloat (`expression`)`

Selects a random float from the specified range [min,max].

Attribute : `min [1]:` floatOrTemplateRef = 0

Attribute : `max [1]:` floatOrTemplateRef

15.2. Expressions Used only in Outcomes Processing

This section describes a number of built-in expressions which can only be used in outcomeProcessing. They return information about sets of items referred to in an assessmentTest.

Abstract class : `itemSubset`

Derived classes:
`numberCorrect`, `numberIncorrect`, `numberPresented`, `numberResponded`, `numberSelected`, `outcomeMaximum`, `outcomeMinimum`, `testVariables`

This class defines the concept of a sub-set of the items selected in an assessmentTest. The attributes define criteria that must be matched by **all** members of the sub-set. It is used to control a number of expressions in outcomeProcessing for returning information about the test as a whole, or arbitrary subsets of it.

Attribute : `sectionIdentifier [0..1]:` identifier
If specified, only variables from items in the assessmentSection with matching identifier are matched. Items in sub-sections are *included* in this definition.

Attribute : `includeCategory [*]:` identifier
If specified, only variables from items with a matching category are included.

Attribute : `excludeCategory [*]:` identifier
If specified, only variables from items with no matching category are included.

Class : `testVariables (`expression, itemSubset`)`

This expression, which can only be used in outcomes processing, simultaneously looks up the value of an itemVariable in a sub-set of the items referred to in a test. Only variables with single cardinality are considered, all NULL values are **ignored**. The result has cardinality multiple and base-type as specified below.

Attribute : `variableIdentifier [1]:` identifier
The identifier of the variable to look up in each item. If a test brings together items with different variable naming conventions variableMappings may be used to reduce the complexity of

outcomes processing and allow a single testVariables expression to be used. Items with no matching variable are ignored.

Attribute : `baseType [0..1]:` baseType
If specified, matches only variables declared with this baseType. This also becomes the base-type of the result (subject to type promotion through weighting, as described below). If omitted, variables declared with base-type integer *or* float are matched. The base-type of the result is integer if *all* matching values have base-type integer, otherwise it is float and integer values are subject to type promotion.

Attribute : `weightIdentifier [0..1]:` identifier
If specified, the defined weight is applied to each variable as described in the definition of weightIdentifier for a single variable. **The behavior of this attribute is only defined if baseType is float or omitted**. When a weighting is specified the result of the expression always has base-type float.

Class : `outcomeMaximum (`expression, itemSubset)

This expression, which can only be used in outcomes processing, simultaneously looks up the normalMaximum value of an *outcome* variable in a sub-set of the items referred to in a test. Only variables with single cardinality are considered. If any of the items within the given subset have no declared maximum the result is NULL, otherwise the result has cardinality multiple and base-type float.

Attribute : `outcomeIdentifier [1]:` identifier
As per the variableIdentifier attribute of testVariables.

Attribute : `weightIdentifier [0..1]:` identifier
As per the weightIdentifier attribute of testVariables.

Class : `outcomeMinimum (`expression, itemSubset)

This expression, which can only be used in outcomes processing, simultaneously looks up the normalMinimum value of an *outcome* variable in a sub-set of the items referred to in a test. Only variables with single cardinality are considered. Items with no declared minimum are **ignored**. The result has cardinality multiple and base-type float.

Attribute : `outcomeIdentifier [1]:` identifier
As per the variableIdentifier attribute of testVariables.

Attribute : `weightIdentifier [0..1]:` identifier
As per the weightIdentifier attribute of testVariables.

Class : `numberCorrect (`expression, itemSubset)

This expression, which can only be used in outcomes processing, calculates the number of items in a given sub-set, for which the all defined response variables match their associated correctResponse. Only items for which **all** declared response variables have correct responses defined are considered. The result is an integer with single cardinality.

Class : `numberIncorrect (`expression, itemSubset)

This expression, which can only be used in outcomes processing, calculates the number of items in a given sub-set, for which at least one of the defined response variables **does not** match its associated correctResponse. Only items for which **all** declared response variables have correct responses defined *and have been attempted at least once* are considered. The result is an integer with single cardinality.

Class : `numberResponded (`expression, itemSubset)

This expression, which can only be used in outcomes processing, calculates the number of items in a given sub-set that have been *attempted* (at least once) and for which a response was given. In other words, items for which at least one declared response has a value that differs from its declared default (typically NULL). The result is an integer with single cardinality.

Class : `numberPresented (`expression, itemSubset)

This expression, which can only be used in outcomes processing, calculates the number of items in a given sub-set that have been *attempted* (at least once). In other words, items with which the user has interacted, whether or not they provided a response. The result is an integer with single cardinality.

Class : `numberSelected (`expression, itemSubset)

This expression, which can only be used in outcomes processing, calculates the number of items in a given sub-set that have been selected for presentation to the candidate, regardless of whether the candidate has attempted them or not. The result is an integer with single cardinality.

15.3. Operators

Operators are a family of classes derived from expression that obtain their value (referred to as their result) either by modifying a single sub-expression or by combining two or more sub-expressions in a specified way. Operators never effect the values of itemVariables directly, in other words, there are no 'side effects'.

All operators have a baseType and cardinality though these may be dependent on the sub-expression(s) they contain.

Class : `multiple (`expression)

Contains : expression [*]

The multiple operator takes 0 or more sub-expressions all of which must have either single or multiple cardinality. Although the sub-expressions may be of any base-type they must all be of *the same* base-type. The result is a container with multiple cardinality containing the values of the sub-expressions, sub-expressions with multiple cardinality have their individual values added to the result: *containers cannot contain other containers*. For example, when applied to A, B and {C,D} the multiple operator results in {A,B,C,D}. All sub-expressions with NULL values are ignored. If no sub-expressions are given (or all are NULL) then the result is NULL.

Class : `ordered (`expression)

Contains : expression [*]

The ordered operator takes 0 or more sub-expressions all of which must have either single or ordered cardinality. Although the sub-expressions may be of any base-type they must all be of *the same* base-type. The result is a container with ordered cardinality containing the values of the sub-expressions, sub-expressions with ordered cardinality have their individual values added (in order) to the result: *contains cannot contain other containers*. For example, when applied to A, B, {C,D} the ordered operator results in {A,B,C,D}. Note that the ordered operator never results in an empty container. All sub-expressions with NULL values are ignored. If no sub-expressions are given (or all are NULL) then the result is NULL

Class : `containerSize (`expression)

Contains : expression [1]

The containerSize operator takes a sub-expression with any base-type and either multiple or ordered cardinality. The result is an integer giving the number of values in the sub-expression, in other words, the size of the container. *If the sub-expression is NULL the result is 0.* This operator can be used for determining how many choices were selected in a multiple-response choiceInteraction, for example.

Class : `isNull (`expression)

Contains : expression [1]

The isNull operator takes a sub-expression with any base-type and cardinality. The result is a single boolean with a value of true if the sub-expression is NULL and false otherwise. Note that empty containers and empty strings are both treated as NULL.

Class : `index (`expression)

Attribute : `n [1]:` integer

Contains : expression [1]

The index operator takes a sub-expression with an ordered container value and any base-type. The result is the n^{th} value of the container. The result has the same base-type as the sub-expression but single cardinality. The first value of a container has index 1, the second 2 and so on. n must be a positive integer. If n exceeds the number of values in the container then the result of the index operator is NULL.

Class : `fieldValue (`expression`)`

Attribute : `fieldIdentifier [1]:` identifier
The identifier of the field to be selected.

Contains : expression [1]

The field-value operator takes a sub-expression with a record container value. The result is the value of the field with the specified fieldIdentifier. If there is no field with that identifier then the result of the operator is NULL.

Class : `random (`expression`)`

Contains : expression [1]

The random operator takes a sub-expression with a multiple or ordered container value and any base-type. The result is a single value randomly selected from the container. The result has the same base-type as the sub-expression but single cardinality. If the sub-expression is NULL then the result is also NULL.

Class : `member (`expression`)`

Contains : expression [2]

The member operator takes two sub-expressions which must both have the same base-type. The first sub-expression must have single cardinality and the second must be a multiple or ordered container. The result is a single boolean with a value of true if the value given by the first sub-expression is in the container defined by the second sub-expression. If either sub-expression is NULL then the result of the operator is NULL.

The member operator should not be used on sub-expressions with a base-type of float because of the poorly defined comparison of values. It **must** not be used on sub-expressions with a base-type of duration.

Class : `delete (`expression`)`

Contains : expression [2]

The delete operator takes two sub-expressions which must both have the same base-type. The first sub-expression must have single cardinality and the second must be a multiple or ordered

container. The result is a new container derived from the second sub-expression with *all* instances of the first sub-expression removed. For example, when applied to A and {B,A,C,A} the result is the container {B,C}. If either sub-expression is NULL the result of the operator is NULL.

The restrictions that apply to the member operator also apply to the delete operator.

Class : `contains (`expression)

Contains : expression [2]

The contains operator takes two sub-expressions which must both have the same base-type and cardinality—either multiple or ordered. The result is a single boolean with a value of true if the container given by the first sub-expression contains the value given by the second sub-expression and false if it doesn't. Note that the contains operator works differently depending on the cardinality of the two sub-expressions. For unordered containers the values are compared without regard for ordering, for example, [A,B,C] contains [C,A]. Note that [A,B,C] does not contain [B,B] but that [A,B,B,C] does. For ordered containers the second sub-expression must be a strict sub-sequence within the first. In other words, [A,B,C] does not contain [C,A] but it does contain [B,C].

If either sub-expression is NULL then the result of the operator is NULL. Like the member operator, the contains operator should not be used on sub-expressions with a base-type of float and **must** not be used on sub-expressions with a base-type of duration.

Class : `substring (`expression)

Contains : expression [2]

The substring operator takes two sub-expressions which must both have an effective base-type of string and single cardinality. The result is a single boolean with a value of true if the first expression is a substring of the second expression and false if it isn't. If either sub-expression is NULL then the result of the operator is NULL.

Attribute : `caseSensitive [1]:` boolean = true
Used to control whether or not the substring is matched case sensitively. If true then the match is case sensitive and, for example, "Hell" is not a substring of "Shell". If false then the match is not case sensitive and "Hell" *is* a substring of "Shell".

Class : `not (`expression)

Contains : expression [1]

The not operator takes a single sub-expression with a base-type of boolean and single cardinality. The result is a single boolean with a value obtained by the logical negation of the sub-expression's value. If the sub-expression is NULL then the not operator also results in NULL.

Class : `and (`expression)

Contains : expression [1..*]

The and operator takes one or more sub-expressions each with a base-type of boolean and single cardinality. The result is a single boolean which is true if all sub-expressions are true and false if any of them are false. If one or more sub-expressions are NULL and all others are true then the operator also results in NULL.

Class : `or (`expression)

Contains : expression [1..*]

The or operator takes one or more sub-expressions each with a base-type of boolean and single cardinality. The result is a single boolean which is true if any of the sub-expressions are true and false if all of them are false. If one or more sub-expressions are NULL and all the others are false then the operator also results in NULL.

Class : `anyN (`expression)

Contains : expression [1..*]

The anyN operator takes one or more sub-expressions each with a base-type of boolean and single cardinality. The result is a single boolean which is true if at least min of the sub-expressions are true and at most max of the sub-expressions are true. If more than n - min sub-expressions are false (where n is the total number of sub-expressions) or more than max sub-expressions are true then the result is false. If one or more sub-expressions are NULL then it is possible that neither of these conditions is satisfied, in which case the operator results in NULL. For example, if min is 3 and max is 4 and the sub-expressions have values {true,true,false,NULL} then the operator results in NULL whereas {true,false,false,NULL} results in false and {true,true,true,NULL} results in true. The result NULL indicates that the correct value for the operator cannot be determined.

Attribute : `min [1]:` integerOrTemplateRef
The minimum number of sub-expressions that must be true.

Attribute : `max [1]:` integerOrTemplateRef
The maximum number of sub-expressions that may be true.

Class : `match (`expression)

Contains : expression [2]

The match operator takes two sub-expressions which must both have the same base-type and cardinality. The result is a single boolean with a value of true if the two expressions represent the

same value and false if they do not. If either sub-expression is NULL then the operator results in NULL.

The match operator must not be confused with broader notions of equality such as numerical equality. To avoid confusion, the match operator should not be used to compare subexpressions with base-types of float and **must** not be used on sub-expressions with a base-type of duration.

Class : `stringMatch (`expression`)`

Contains : expression [2]

The stringMatch operator takes two sub-expressions which must have single and a base-type of string. The result is a single boolean with a value of true if the two strings match according to the comparison rules defined by the attributes below and false if they don't. If either sub-expression is NULL then the operator results in NULL.

Attribute : `caseSensitive [1]:` boolean
Whether or not the match is to be carried out case sensitively.

Attribute : `substring [0..1]:` boolean = false
This attribute is now deprecated, the substring operator should be used instead. If true, then the comparison returns true if the first string *contains* the second one, otherwise it returns true only if they match entirely.

Class : `patternMatch (`expression`)`

Contains : expression [1]

The patternMatch operator takes a sub-expression which must have single cardinality and a base-type of string. The result is a single boolean with a value of true if the sub-expression matches the regular expression given by pattern and false if it doesn't. If the sub-expression is NULL then the operator results in NULL.

Attribute : `pattern [1]:` stringOrTemplateRef
The syntax for the regular expression language is as defined in Appendix F of [XML_SCHEMA2].

Class : `equal (`expression`)`

Contains : expression [2]

The equal operator takes two sub-expressions which must both have single cardinality and have a numerical base-type. The result is a single boolean with a value of true if the two expressions are numerically equal and false if they are not. If either sub-expression is NULL then the operator results in NULL.

Attribute : `toleranceMode [1]:` toleranceMode = exact
When comparing two floating point numbers for equality it is often desirable to have a tolerance to ensure that spurious errors in scoring are not introduced by rounding errors. The tolerance mode determines whether the comparison is done exactly, using an absolute range or a relative range.

Attribute : `tolerance [0..2]:` floatOrTemplateRef
If the tolerance mode is absolute or relative then the tolerance must be specified. The tolerance consists of two positive numbers, *t0* and *t1*, that define the lower and upper bounds. If only one value is given it is used for both.

In absolute mode the result of the comparison is true if the value of the second expression, *y* is within the following range defined by the first value, *x*.

```
x-t0,x+t1
```

In relative mode, *t0* and *t1* are treated as percentages and the following range is used instead.

```
x*(1-t0/100),x*(1+t1/100)
```

Attribute : `includeLowerBound [0..1]:` boolean = true
Controls whether or not the lower bound is included in the comparison

Attribute : `includeUpperBound [0..1]:` boolean = true
Controls whether or not the upper bound is included in the comparison

Enumeration: `toleranceMode`

```
exact

absolute

relative
```

Class : `equalRounded (`expression**)**

Contains : expression [2]

The equalRounded operator takes two sub-expressions which must both have single cardinality and have a numerical base-type. The result is a single boolean with a value of true if the two expressions are numerically equal after rounding and false if they are not. If either sub-expression is NULL then the operator results in NULL.

Attribute : `roundingMode [1]:` roundingMode = significantFigures
Numbers are rounded to a given number of significantFigures or decimalPlaces.

Attribute : `figures [1]:` integerOrTemplateRef
The number of figures to round to.

For example, if the two values are 1.56 and 1.6 and significantFigures mode is used with figures=2 then the result would be true.

Enumeration: `roundingMode`

```
significantFigures

decimalPlaces
```

Class : `inside (`expression)

Contains : expression [1]

The inside operator takes a single sub-expression which must have a baseType of point. The result is a single boolean with a value of true if the given point is inside the area defined by shape and coords. If the sub-expression is a container the result is true if *any* of the points are inside the area. If either sub-expression is NULL then the operator results in NULL.

Attribute : `shape [1]:` shape
The shape of the area.

Attribute : `coords [1]:` coords
The size and position of the area, interpreted in conjunction with the shape.

Class : `lt (`expression)

Contains : expression [2]

The lt operator takes two sub-expressions which must both have single cardinality and have a numerical base-type. The result is a single boolean with a value of true if the first expression is numerically less than the second and false if it is greater than or equal to the second. If either sub-expression is NULL then the operator results in NULL.

Class : `gt (`expression)

Contains : expression [2]

The gt operator takes two sub-expressions which must both have single cardinality and have a numerical base-type. The result is a single boolean with a value of true if the first expression is numerically greater than the second and false if it is less than or equal to the second. If either sub-expression is NULL then the operator results in NULL.

Class : `lte (`expression)

Contains : expression [2]

The lte operator takes two sub-expressions which must both have single cardinality and have a numerical base-type. The result is a single boolean with a value of true if the first expression is numerically less than or equal to the second and false if it is greater than the second. If either sub-expression is NULL then the operator results in NULL.

Class : `gte` (expression)

Contains : expression [2]

The gte operator takes two sub-expressions which must both have single cardinality and have a numerical base-type. The result is a single boolean with a value of true if the first expression is numerically less than or equal to the second and false if it is greater than the second. If either sub-expression is NULL then the operator results in NULL.

Class : `durationLT` (expression)

Contains : expression [2]

The durationLT operator takes two sub-expressions which must both have single cardinality and base-type duration. The result is a single boolean with a value of true if the first duration is shorter than the second and false if it is longer than (or equal) to the second. If either sub-expression is NULL then the operator results in NULL.

There is no 'durationLTE' or 'durationGT' because equality of duration is meaningless given the variable precision allowed by duration. Given that duration values are obtained by truncation rather than rounding it makes sense to test only less-than or greater-than-equal inequalities only. For example, if we want to determine if a candidate took less than 10 seconds to complete a task in a system that reports durations to a resolution of *epsilon* seconds (*epsilon*<1) then a value equal to 10 would cover all durations in the range [10,10+epsilon).

Class : `durationGTE` (expression)

Contains : expression [2]

The durationGTE operator takes two sub-expressions which must both have single cardinality and base-type duration. The result is a single boolean with a value of true if the first duration is longer (or equal, within the limits imposed by truncation as described above) than the second and false if it is shorter than the second. If either sub-expression is NULL then the operator results in NULL.

See durationLT for more information about testing the equality of durations.

Class : `sum` (expression)

Contains : expression [1..*]

The sum operator takes 1 or more sub-expressions which all have single cardinality and have numerical base-types. The result is a single float or, if all sub-expressions are of integer type, a single integer that corresponds to the sum of the numerical values of the sub-expressions. If any of the sub-expressions are NULL then the operator results in NULL.

Class : `product (`expression)

Contains : expression [1..*]

The product operator takes 1 or more sub-expressions which all have single cardinality and have numerical base-types. The result is a single float or, if all sub-expressions are of integer type, a single integer that corresponds to the product of the numerical values of the sub-expressions. If any of the sub-expressions are NULL then the operator results in NULL.

Class : `subtract (`expression)

Contains : expression [2]

The subtract operator takes 2 sub-expressions which all have single cardinality and numerical base-types. The result is a single float or, if both sub-expressions are of integer type, a single integer that corresponds to the first value minus the second. If either of the sub-expressions is NULL then the operator results in NULL.

Class : `divide (`expression)

Contains : expression [2]

The divide operator takes 2 sub-expressions which both have single cardinality and numerical base-types. The result is a single float that corresponds to the first expression divided by the second expression. If either of the sub-expressions is NULL then the operator results in NULL.

Item authors should make every effort to ensure that the value of the second expression is never 0, however, if it is zero or the resulting value is outside the value set defined by float (not including positive and negative infinity) then the operator should result in NULL.

Class : `power (`expression)

Contains : expression [2]

The power operator takes 2 sub-expression which both have single cardinality and numerical base-types. The result is a single float that corresponds to the first expression raised to the power of the second. If either or the sub-expressions is NULL then the operator results in NULL.

If the resulting value is outside the value set defined by float (not including positive and negative infinity) then the operator shall result in NULL.

Class : `integerDivide (`expression)

Contains : expression [2]

The integer divide operator takes 2 sub-expressions which both have single cardinality and base-type integer. The result is the single integer that corresponds to the first expression (x) divided by the second expression (y) rounded down to the greatest integer (i) such that i<=(x/y). If y is 0, or if either of the sub-expressions is NULL then the operator results in NULL.

Class : `integerModulus (`expression)

Contains : expression [2]

The integer modulus operator takes 2 sub-expressions which both have single cardinality and base-type integer. The result is the single integer that corresponds to the remainder when the first expression (x) is divided by the second expression (y). If z is the result of the corresponding integerDivide operator then the result is x-z*y. If y is 0, or if either of the sub-expressions is NULL then the operator results in NULL.

Class : `truncate (`expression)

Contains : expression [1]

The truncate operator takes a single sub-expression which must have single cardinality and base-type float. The result is a value of base-type integer formed by truncating the value of the sub-expression towards zero. For example, the value 6.8 becomes 6 and the value -6.8 becomes -6. If the sub-expression is NULL then the operator results in NULL.

Class : `round (`expression)

Contains : expression [1]

The round operator takes a single sub-expression which must have single cardinality and base-type float. The result is a value of base-type integer formed by rounding the value of the sub-expression. The result is the integer *n* for all input values in the range [n-0.5,n+0.5). In other words, 6.8 and 6.5 both round up to 7, 6.49 rounds down to 6 and -6.5 rounds up to **-6**. If the sub-expression is NULL then the operator results in NULL.

Class : `integerToFloat (`expression)

Contains : expression [1]

The integer to float conversion operator takes a single sub-expression which must have single cardinality and base-type integer. The result is a value of base type float with the same numeric value. If the sub-expression is NULL then the operator results in NULL.

Class : `customOperator (`expression**)**

The custom operator provides an extension mechanism for defining operations not currently supported by this specification.

Attribute : `class [0..1]:` identifier
The class attribute allows simple sub-classes to be named. The definition of a sub-class is tool specific and may be inferred from toolName and toolVersion.

Attribute : `definition [0..1]:` uri
A URI that identifies the definition of the custom operator in the global namespace.

In addition to the class and definition attributes, sub-classes may add any number of attributes of their own.

Contains : expression [*]
Custom operators can take any number of sub-expressions of any type to be treated as parameters.

It has been suggested that customOperator might be used to help link processing rules defined by this specification to instances of web-service based processing engines. For example, a web-service which offered automated marking of free text responses. Implementors experimenting with this approach are encouraged to share information about their solutions to help determine the best way to achieve this type of processing.

16. Item and Test Fragments

An Item Fragment is part of an item that is managed independently of the items that depend on it. Similarly, a Test Fragment is part of a test that is managed independently of the tests that depend on it. Fragments are packaged as separate resources and can be transported independently. A fragment is defined as an instance of any class descended from one of the following abstract classes: flow, responseRule, sectionPart or outcomeRule. For example, an item fragment may be a division of the itemBody represented by an instance of the div class.

Class : `include` (flow, outcomeRule, responseRule, sectionPart)

Fragments are included using the *Xinclude* mechanism. (See [XINCLUDE].) The instance of *include* is treated as if it was actually an instance of the root element of the fragment referred to by the *href* attribute of the include element. For the purposes of this specification the xpointer mechanism defined by the XInclude specification must not be used. Also, all included fragments must be treated as parsed xml.

This technique is similar to the inclusion of media objects (using object) but allows the inclusion of data that conforms to this specification, specifically, it allows the inclusion of interactions, static content, processing rules or, at test level whole sections, to be included from externally defined fragments.

When including externally defined fragments the content of the fragment **must** satisfy the requirements of the specification in the context of the item in which it is being included. For example, interactions included from fragments must be correctly bound to response variables declared in the items.

Class : `responseProcessingFragment` (responseRule)

Contains : responseRule [*]

A responseProcessingFragment is a simple group of responseRules which are grouped together in order to allow them to be managed as a separate resource. It should not be used for any other purpose.

Note that a response processing *template* allows a system to carry out response processing *without* having to parse the individual response processing rules. On the other hand, a responseProcessing element containing a reference to a an externally defined response processing fragment *must* be parsed to determine the actions to carry out.

Class : `outcomeProcessingFragment` (outcomeRule)

Contains : outcomeRule [*]

An outcomeProcessingFragment is a simple group of outcomeRules which are grouped together in order to allow them to be managed as a separate resource. It should not be used for any other purpose.

17. Basic Data Types

Datatype: `boolean`

A boolean value is either true or false. Note that lexical bindings to strings such as "Yes", "TRUE", "1", etc. are outside the scope of this document.

Datatype: `coords`

The coords type provides the coordinates that determine the size and location of an area defined by a corresponding shape.

The coordinates themselves are an ordered list of lengths (as defined in [XHTML]). The interpretation of each length value is dependent on the value of the associated shape as follows.

- rect: left-x, top-y, right-x, bottom-y.
- circle: center-x, center-y, radius. Note. When the radius value is a percentage value, user agents should calculate the final radius value based on the associated object's width and height. The radius should be the smaller value of the two.
- poly: x1, y1, x2, y2, ..., xN, yN. The first x and y coordinate pair and the last should be the same to close the polygon. When these coordinate values are not the same, user agents should infer an additional coordinate pair to close the polygon.
- ellipse: center-x, center-y, h-radius, v-radius. Note that the ellipse shape is deprecated as it is not defined by [XHTML].
- default: no coordinates should be given.

Datatype: `date`

A fully-specified calendar date, including year, month and day of month from the reference system defined in [ISO8601]. Valid years range from 0001-9999. Year values have no timezone information.

Datatype: `datetime`

A fully-specified calendar date and time from the reference system defined in [ISO8601]. Valid years range from 0000-9999.

Datatype: `duration`

A period of time, measured in seconds.

Datatype: `float`

The IEEE double-precision 64-bit floating point type.

Datatype: `identifier`

An identifier is simply a logical reference to another object in the item, such as an itemVariable or choice. An identifier is a string of characters that must start with a Letter or an underscore ('_') and contain only Letters, underscores, hyphens ('-'), period ('.', a.k.a. full-stop), Digits, CombiningChars and Extenders. Identifiers containing the period character are reserved for future use. The character classes Letter, Digit, CombiningChar and Extender are defined in the Extensible Markup Language (XML) 1.0 (Second Edition) [XML]. Note particularly that identifiers may not contain the colon (':') character. Identifiers should have no more than 32 characters for compatibility with version 1. They are always compared case-sensitively.

Datatype: `integer`

An integer value is a whole number in the range [-2147483648,2147483647]. This is the range of a twos-complement 32-bit integer.

Datatype: `language`

Natural language identifiers as defined by [RFC3066].

Datatype: `length`

The length datatype is as defined in [XHTML].

Datatype: `mimeType`

The set of mime types (type and subtype), as defined by [RFC2045].

Enumeration: `orientation`

`vertical`

`horizontal`

Enumeration: `shape`

A value of a shape is alway accompanied by coordinates (see coords and an associated image which provides a context for interpreting them.

`default`
The default shape refers to the entire area of the associated image.

`rect`
A rectangular region.

`circle`
A circular region

`poly`
An arbitrary polygonal region

`ellipse`
This value is deprecated, but is included for compatibility with version of 1 of the QTI specification. Systems should use circle or poly shapes instead.

Datatype: `string`

A string value is any sequence of characters. A character is anything in the class Char defined in Extensible Markup Language (XML) 1.0 (Second Edition).

Datatype: `string256`

A string value (as per string) that is limited to 256 characters in length.

Datatype: `styleclass`

The type used when referring to a class definition, for example in a stylesheet. Class names cannot contain spaces.

Datatype: `uri`

A Uniform Resource Identifier as defined in [URI].

Datatype: `valueType`

A simple type used to represent a single value of any baseType as defined by the assessment variable data model.

Enumeration: `view`

`author`

`candidate`

`proctor`
Sometimes referred to as an *invigilator*

`scorer`

`testConstructor`

`tutor`

About This Document

Title	IMS Question and Test Interoperability Assessment Test, Section, and Item Information Model
Editors	Steve Lay (Cambridge Assessment), Pierre Gorissen (SURF)
Version	Public Draft Draft v2.1
Version Date	8 June 2006
Status	**Public Draft Specification**
Summary	This document describes the QTI Assessment Test, Section, and Item Information Model specification.
Revision Information	8 June 2006
Purpose	This document has been approved by the IMS Technical Board and is made available for public review and comment.
Document Location	

List of Contributors

The following individuals contributed to the development of this document:

Name	Organization
Dick Bacon	University of Surrey
Niall Barr	Question Mark
Lance Blackstone	Pearson VUE
Jeanne Ferrante	ETS
Pierre Gorissen	SURF
Regina Hoag	ETS
Gopal Krishnan	Pearson VUE
Steve Lay	Cambridge Assessment
Rowin Young	CETIS

Revision History

Version No.	Release Date	Comments
Base Document 2.1	14 October 2005	The first version of the QTI v2.1 specification.
Public Draft 2.1	9 January 2006	The Public Draft v2.1 of the QTI specification.
Public Draft 2.1 (revision 2)	8 June 2006	The Public Draft v2.1 (revision 2) of the QTI specification.