IMS Question and Test Interoperability Information Model

Version 2.0 Final Specification

Document Name: IMS Question and Test Interoperability Information Model
Revision: 24 January 2005

Date Issued:	24 January 2005
Latest version:	http://www.imsglobal.org/question/qti_v2p0/imsqti_infov2p0.html
Supersedes:	QTI Item v2.0 Public Draft specification, 07 June 2004, http://www.imsglobal.org/question/
Register comments or implementations:	http://www.imsglobal.org/developers/ims/imsforum/categories.cfm?catid=23

IMS Global Learning Consortium has made no inquiry into whether or not the implementation of third party material included in this specification would infringe upon the intellectual property rights of any party.

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.

Table of Contents

1. Introduction

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)

RR
IMS Question & Test Interoperability: Results Reporting Specification, Version 1.2
Published: 2002-02

URI
RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax
Published: 1998-08

XHTML
XHTML 1.1: The Extensible HyperText Markup Language

XHTML_MOD
XHTML Modularization
http://www.w3.org/MarkUp/modularization

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

Adaptive Tests are out of scope for this specification.

Assessment

An Assessment is equivalent to a 'Test'. It contains the collection of Items that are used to determine the level of mastery, or otherwise, that a participant has on a particular subject. The Assessment contains all of the necessary instructions to enable variable sequencing of the Items and the corresponding aggregated scoring to produce the final test score. Assessments are out of scope for this document.

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.

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 and Modal 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 Session

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

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 a contains 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. See also the class itemVariable.

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.

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. For an Item, an outcome is represented by one or more Outcome Variables.

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. See also the class outcomeVariable.

Response Processing

The process by which the values of Response Variables are judged (scored) and the values of Outcome Variables 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 responseVariable

Scoring Engine

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

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.

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 record the amount of time spent by the Candidate completing it. In practice, this information may be collected by a Delivery Engine but it is not used for Response Processing and the method by which it is reported is outside the scope of this specification.

4. Items

Class : `assessmentItem`

Attribute : `identifier [1]:` string

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 [*]

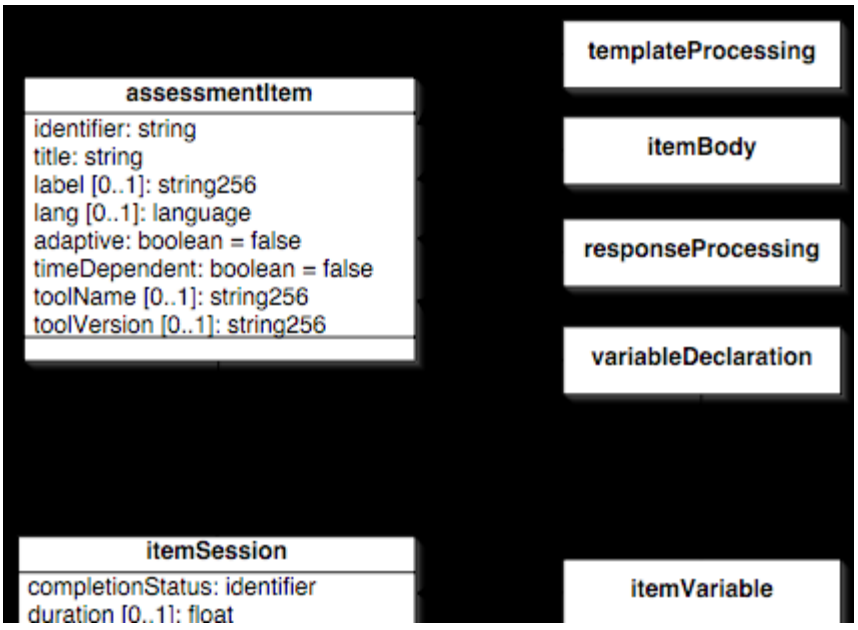
Contains : templateProcessing [0..1]

Contains : stylesheet [0..*]

Contains : itemBody [0..1]

Contains : responseProcessing [0..1]

Contains : modalFeedback [*]



Item Sessions

Abstract class : `itemSession`

itemSession is an abstract class to help illustrate the requirements on Delivery Engines when delivering to candidates items that conform to this specification.

Associated with : assessmentItem [1]
An itemSession is associated with one and only one assessmentItem.

Attribute : `completionStatus [1]:` identifier
Delivery Engines must maintain the value of the built-in outcome variable *completionStatus* as part of the session state. It starts with the reserved value "not_attempted". At the start of the first attempt it changes the 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 *complete* then the session must be placed into the closed state, however, an itemSession is **not** required to wait for the complete 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. Similarly, Non-adaptive Items are not *required* to set a value for completionStatus, however, Adaptive Items *must* maintain a suitable value and should set completionStatus to "complete" to indicate when the cycle of interaction, response processing and feedback must stop. Delivery Engines are encouraged to use the value of completionStatus when communicating using [CMI]. See the accompanying integration guide for more details.

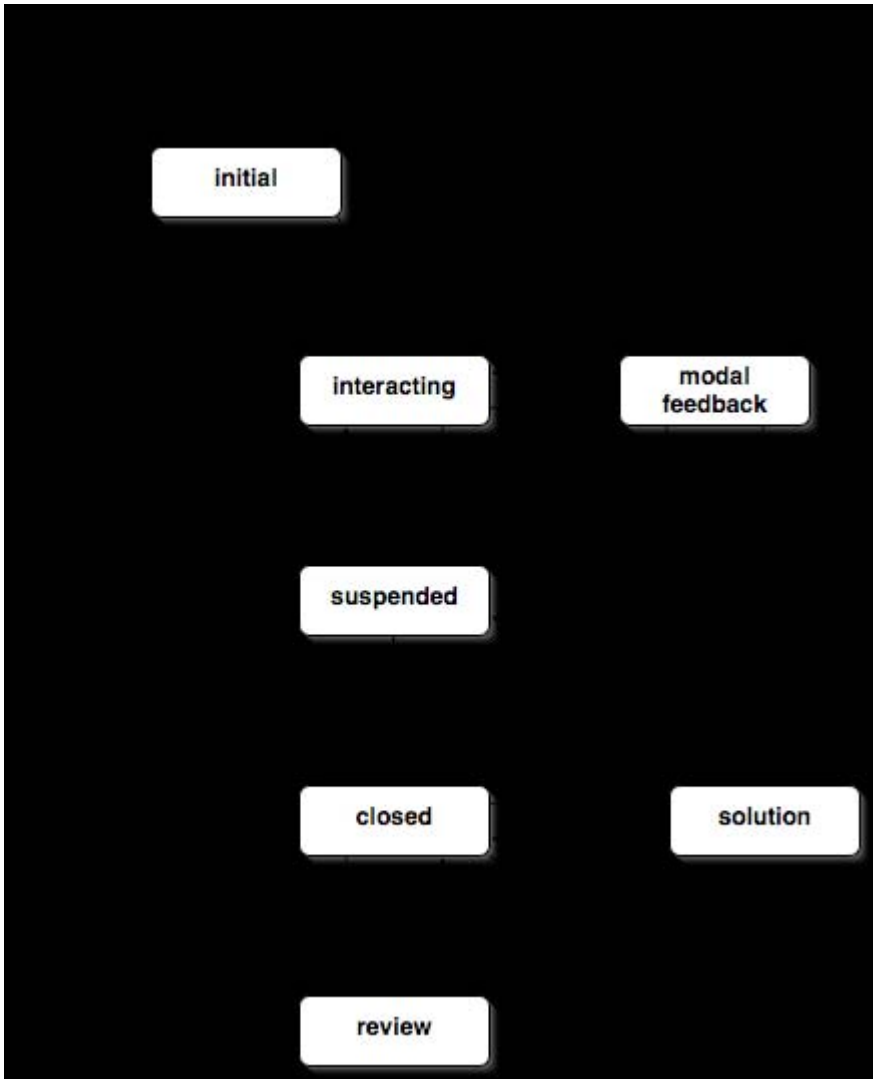
Attribute : `duration [0..1]:` float
Systems that support Time Dependent Items must record the duration of the session. The duration is defined as being the accumulated time (in seconds) of all Candidate Sessions for all Attempts. In other words the time between the beginning and the end of the itemSession **minus** any time the itemSession 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,duration+*epsilon*). In other words, duration values are truncated. For items that are not time dependent duration must not be used.

Contains : itemVariable [*]
The itemSession keeps track of the current values assigned to all itemVariables. The values of completionStatus and duration are treated as special item variables. They share the same namespace as the item variables explicitly declared through variableDeclarations.

Contains : sessionContext [1]
An itemSession is also associated with a sessionContext which provides information about the candidate, when and where the session took place and so on.

The following diagram illustrates the user-perceived states of the itemSession. 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 candidate 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.



Lifecycle of an Item Session

A delivery system notionally creates an instance of an itemSession object when it first becomes eligible for delivery to the candidate. The itemSession's state is then maintained and updated in response to the actions of the candidate until the session is over. At this point the state of the session is turned into a session report (or thrown away). A delivery system may also allow a session report from a past session to be used to re-create the 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).

The initial state of an itemSession 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, itemSessions 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 is placed into the suspended state ready for the next candidate session. During a candidate session the *itemSession* 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 may only be invoked a limited number of times, typically 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 indicates the end of an attempt. The appearance of the item's body, and whether any modal feedback is shown, is determined by the values of the outcomeVariables.

When no more attempts are allowed the itemSession 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 outcomeVariable.

Finally, for systems that support the display of solutions, the itemSession 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).

Abstract class : `sessionContext`

Associated classes:

`itemSession`

The details of sessionContext will be application specific and are therefore outside the scope of this document. Applications that handle result reports using [RR] should consider the limits of the data model imposed on the context element in that specification.

5. Item Variables

Abstract class : `variableDeclaration`

Derived classes:

`outcomeDeclaration`, `responseDeclaration`, `templateDeclaration`

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. The purpose of the declaration is to associate an identifier with the variable and to identify the runtime type of the variable's value. At runtime (i.e., during an itemSession) the value of the variable is notionally represented by a class derived from itemVariable

Attribute : `identifier [1]:` identifier
The identifiers of the built-in session variables are reserved. They are completionStatus 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.

Abstract class : `itemVariable`

Derived classes:
`outcomeVariable`, `responseVariable`, `templateVariable`
Associated classes:
`itemSession`

Associated with : variableDeclaration [1]
At runtime, item variables are created in the itemSession each corresponding to a variableDeclaration in the corresponding assessmentItem.

Attribute : `identifier [1]:` identifier
The purpose of an itemVariable is to associate the runtime value of the variable with the variable's identifier and declaration. At runtime the variable has the cardinality and baseType given in the associated declaration

Contains : value [*]
An itemVariable 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 responseVariable will have a NULL value. Empty containers and empty strings are always treated as NULL values.

Class : `value`

Associated classes:
`ordinaryStatistic`, `defaultValue`, `correctResponse`, `itemVariable`

A class that can represent a single value of any baseType in variable declarations. 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 used for specifying the field identifier for a value that forms part of a record.

Attribute : `baseType [0..1]:` baseType
This attribute is 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 responseVariable 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 modeling.

`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 to a single float. 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]:` value

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.

itemSession defines one built-in pre-bound response variable: duration.

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`

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

Abstract class : `responseVariable` (itemVariable)

At runtime, response variables are instantiated as part of an itemSession. 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 itemSession 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 Engine's 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.

5.2. Outcome Variables

Class : `outcomeDeclaration` (variableDeclaration)

Associated classes:

`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.

itemSession defines one built-in outcome variable: completionStatus.

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.

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.

Abstract class : `outcomeVariable` (itemVariable)

Outcome variables are instantiated as part of an itemSession. 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.

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 itemSession 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 responseVariables. 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. 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`, `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.

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, 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)

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:
drawingInteraction, positionObjectInteraction, positionObjectStage, graphicInteraction, 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`

Class : `colgroup` (bodyElement)

Associated classes:
`table`

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 : th [1..*]

Class : `th` (tableCell)

Associated classes:
`tfoot`

Class : `thead` (bodyElement)

Associated classes:
`table`

Contains : tr [1..*]

Class : `tr` (bodyElement)

Associated classes:
`tbody`, `thead`

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 itemSession.

The first method is based on the value of an outcomeVariable.

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 responseVariable.

Enumeration: `showHide`

`show`

`hide`

Class : `feedbackBlock (`feedbackElement, simpleBlock)

Class : `feedbackInline (`feedbackElement, simpleInline)

Class : `rubricBlock (`simpleBlock)

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)

Attribute : `identifier [1]:` identifier
The outcomeVariable or templateVariable that must have been defined and have single cardinality. The values of responseVariables 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 outcomeVariables 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.

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.

Datatype: `styleclass`

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

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 responseVariables. 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 reflects the value of the associated response variable.

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`, `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.

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 responseVariable 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.

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.

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 so implementors should be aware of the issues concerning the use of default values described in the section on responseVariables.

The orderInteraction must be bound to a responseVariable 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 : `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 responseVariable 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.

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 responseVariable 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.

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.

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 responseVariable 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.

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.

Class : gapText (gapChoice)

A simple run of text to be inserted into a gap by the user.

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 responseVariable 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.

Class : inlineChoice (choice)

Associated classes:
inlineChoiceInteraction

A simple run of text to be displayed to the user.

Abstract class : stringInteraction

Derived classes:
extendedTextInteraction, textEntryInteraction

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

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.

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].

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 responseVariables.

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 responseVariable 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 responseVariable 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 : `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.

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 responseVariable, a NULL value indicating that no choices are selected (the usual case).

The hottextInteraction must be bound to a responseVariable 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.

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.

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.

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 spatial 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 responseVariable 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.

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 responseVariable 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.

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 responseVariable 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.

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 responseVariable 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 responseVariable 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 responseVariable 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 centrePoint 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.

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 responseVariable. The currently selected value, if any, *must* be clearly indicated to the candidate .

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 responseVariable 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 responseVariable 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 responseVariable 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.

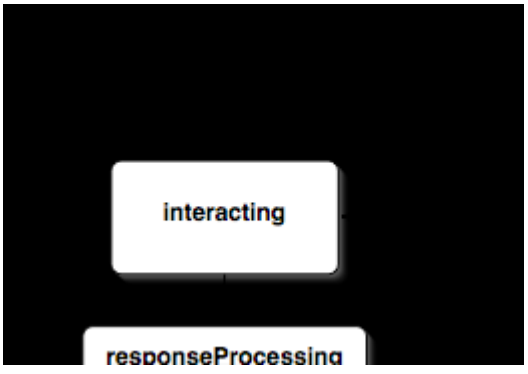
8. Response Processing

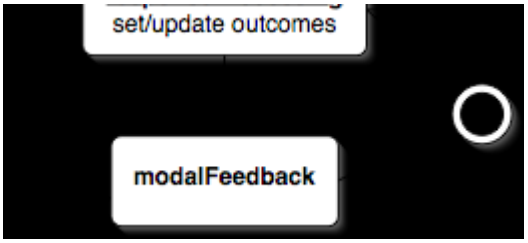
Response processing is the process by which the Delivery Engine assigns outcomes to the itemSession based on the candidate's responses. The outcomes may be used to provide feedback to the candidate. Feedback is either provided immediately following the end of the candidate's attempt or it is provided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

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 itemSession that enters the suspended state may have values for the responseVariables that have yet to be submitted for response processing.

For a Non-adaptive Item the values of the outcomeVariables 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 outcomeVariables 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 *complete*.





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/qti_v2p0/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/qti_v2p0/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*. 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.

Map Response Point
rptemplates/map_response_point.xml

Full template URI:
`http://www.imsglobal.org/question/qti_v2p0/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:

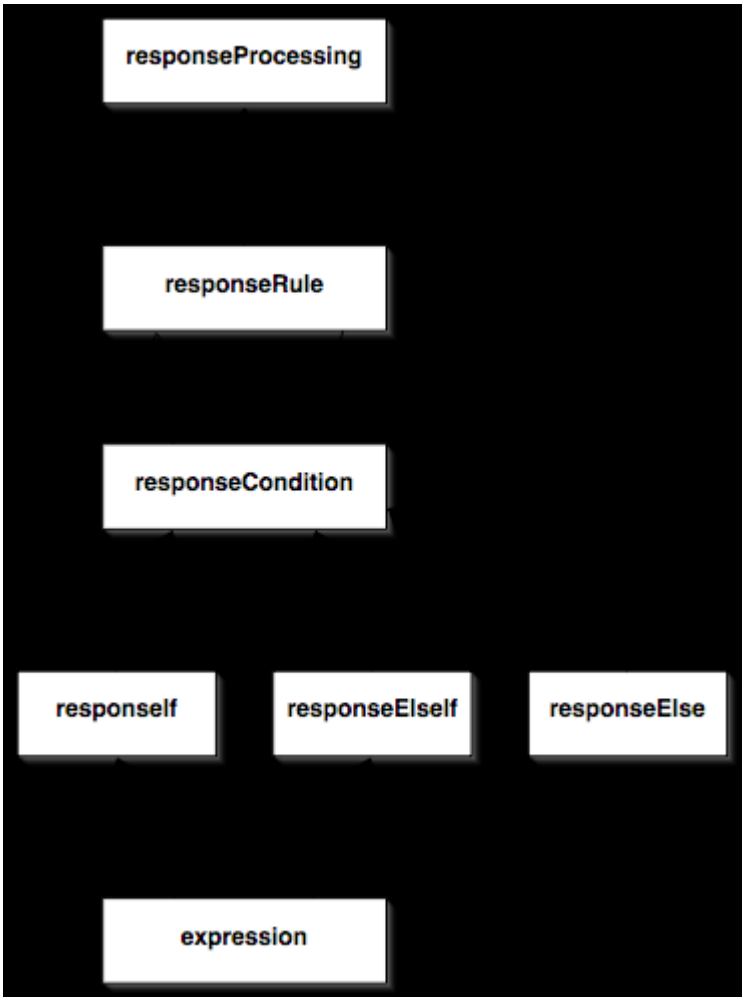
`assessmentItem`

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.

Contains : responseRule [*]

The mapping from values assigned to Response Variables by the candidate onto appropriate values for the item's Outcome Variables is achieved through a number of rules.



Response Processing

Abstract class : `responseRule`

Derived classes:

`exitResponse`, `responseCondition`, `setOutcomeValue`

Associated classes:

`responseElse`, `responseIf`, `responseProcessing`, `responseElseIf`

A response rule is either a responseCondition or a simple action. 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 (`responseRule`)`

Attribute : `identifier [1]:` identifier
The outcomeVariable 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 outcomeVariable being set.

The setOutcomeValue rule sets the value of an outcomeVariable 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 outcomeVariable 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 : `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 outcomeVariable 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. Expressions

Abstract class : `expression`

Derived classes:
`and`, `anyN`, `baseValue`, `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`, `or`, `ordered`, `patternMatch`, `power`, `product`, `random`, `randomFloat`, `randomInteger`, `round`, `stringMatch`, `substring`, `subtract`, `sum`, `truncate`, `variable`
Associated classes:
`and`, `gt`, `ordered`, `divide`, `setCorrectResponse`, `random`, `responseIf`, `substring`,

equalRounded, index, integerDivide, gte, durationLT, contains, durationGTE, member, lt, match, templateIf, product, multiple, power, integerToFloat, setDefaultValue, customOperator, stringMatch, setTemplateValue, setOutcomeValue, not, templateElseIf, integerModulus, subtract, responseElseIf, anyN, round, inside, equal, fieldValue, isNull, patternMatch, lte, sum, truncate, or, 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.

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.

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.

Class : correct (expression)

Attribute : identifier [1]: identifier

This expression looks up the declaration of a responseVariable and returns the associated correctResponse or NULL if no correct value was declared.

Class : mapResponse (expression)

Attribute : identifier [1]: identifier

This expression looks up the value of a responseVariable 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 single or multiple 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 responseVariable 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]:` integer = 0

Attribute : `max [1]:` integer

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

Class : `randomFloat (`expression)

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

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

Attribute : `max [1]:` float

10.1. 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 : `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 nth 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}.

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]:` integer
The minimum number of sub-expressions that must be true.

Attribute : `max [1]:` integer
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 [1]:` boolean = false
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]:` string
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]:` float
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)]
```

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]:` integer
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.

11. 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 itemSession.

Each item cloned from an item template is identical except for the values given to a set of templateVariables. 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 itemSession 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.

Attribute : `paramVariable [1]:` boolean
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 [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.

Abstract class : `templateVariable` (itemVariable)

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

11.1. Using Template Variables in an 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.

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 [*]

11.2. 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 templateVariables. Template processing is identical in form to responseProcessing except that the purpose is to assign values to Template Variables, not outcomeVariables.

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 responseVariable or outcomeVariable. It may only refer to the values of the templateVariables.

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 templateVariable 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 templateVariable being set.

The setTemplateValue rules sets the value of a templateVariable 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 templateVariable being set may appear in the expression where it takes the value previously assigned to it.

Class : `setCorrectResponse` (templateRule)

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

Contains : expression [1]

Class : `setDefaultValue` (templateRule)

Attribute : `identifier [1]:` identifier
The responseVariable or outcomeVariable to have its default value set.

Contains : expression [1]

Class : `exitTemplate` (templateRule)

The exit template rule terminates template processing immediately.

12. 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].

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`

Datatype: `length`

The length datatype is as defined in [XHTML].

Datatype: `mimeType`

Enumeration: `orientation`

`vertical`

`horizontal`

Datatype: `sign`

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`

Datatype: `uri`

A Uniform Resource Identifier as defined in [URI]

Enumeration: `view`

`author`

`candidate`

`proctor`
Sometimes referred to as an *invigilator*

```
scorer

tutor
```

About This Document

Title	IMS Question and Test Interoperability Information Model
Editor	Steve Lay (University of Cambridge)
Version	2.0
Version Date	24 January 2005
Status	**Final Specification**
Summary	This document describes the QTI Information Model specification.
Revision Information	24 January 2005
Purpose	This document has been approved by the IMS Technical Board and is made available for adoption.
Document Location	http://www.imsglobal.org/question/qti_v2p0/imsqti_infov2p0.html

To register any comments or questions about this specification please visit:
http://www.imsglobal.org/developers/ims/imsforum/categories.cfm?catid=23

List of Contributors

The following individuals contributed to the development of this document:

Name	Organization	Name	Organization
Niall Barr	CETIS	Joshua Marks	McGraw-Hill
Sam Easterby-Smith	Canvas Learning	David Poor	McGraw-Hill
Jeanne Ferrante	ETS	Greg Quirus	ETS
Pierre Gorissen	SURF	Niall Sclater	CETIS
Regina Hoag	ETS	Colin Smythe	IMS
Christian Kaefer	McGraw-Hill	GT Springer	Texas Instruments
John Kleeman	Question Mark	Colin Tattersall	OUNL

Steve Lay	UCLES	Rowin Young	CETIS
Jez Lord	Canvas Learning		

Revision History

Version No.	Release Date	Comments
Base Document 2.0	09 March 2004	The first version of the QTI Item v2.0 specification.
Public Draft 2.0	07 June 2004	The Public Draft version 2.0 of the QTI Item Specification.
Final 2.0	24 January 2005	The Final version 2.0 of the QTI specification.