Exercise system in ActiveMath

Alberto González Palomo

2006-06-25

Abstract

This document describes the design and implementation of a part of the e-Learning system ActiveMath, the exercise sub-system, that was done completely by the report author.

The main goal of this system is to facilitate reuse of both the software components and the exercise content by clear and strict separation of functionality.

It consists of a user interface mediator, that handles the interaction with the user (display of questions and feedback, and parsing of the answers), an answer evaluator/diagnoser, that classifies the answer for choosing the adequate feedback, an exercise generator, that produces the exercise content as needed (such as feedback content based on the diagnosis done by the answer evaluator), and an interaction manager that orchestrates all of them.

The exercises are described by an "interaction graph" that is traversed by the interaction manager. The answers from the user are parsed into OpenMath[1] expressions that can be evaluated internally (by the evaluator/diagnoser) or, when required, by an external program such as a Computer Algebra System (CAS) with OpenMath support.

The input parsers implement a variatey of syntaxes similar to those of the Computer Algebra Systems Yacas[2], Maxima[3], Axiom[4], Maple™[5], Mathematica®[6], MuPAD[7], Derive[8], and REDUCE[9]. The intention is that users already familiar with any of those systems can use ActiveMath without having to learn yet another syntax.

[1] A format for mathematical expressions, which specifies unique identifiers for each mathematical concept independently of the notation used for it. `http://www.openmath.org`

[2] `http://yacas.sourceforge.net`

[3] `http://maxima.sourceforge.net`

[4] `http://www.axiom-developer.org`

[5] `http://www.maplesoft.com`

[6] `http://www.mathematica.com`

[7] `http://www.mupad.de`

[8] `http://www.derive.com`

[9] `http://www.reduce-algebra.com`

Abstract

Contents

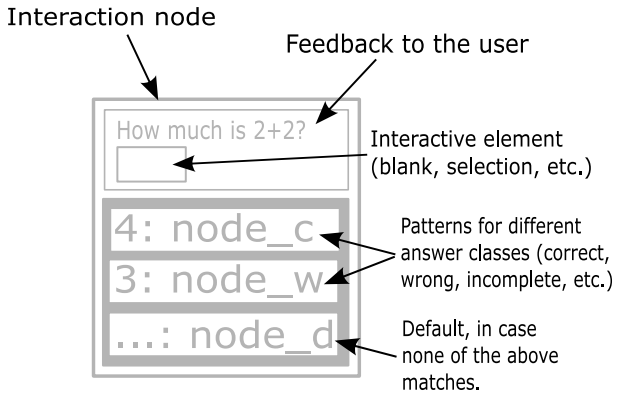
1 Introduction

This document describes the exercise system in the adaptive learning environment ActiveMath. This works has three parts: the exercise representation, the general architecture for a system capable of running exercises so encoded, and the specific implementation done for ActiveMath in the programming language Java. The exercise representation is described in detail in other papers, but we include a brief description here as necessary for a clear understanding of the architecture and implementation.

2 Basic structure of an exercise

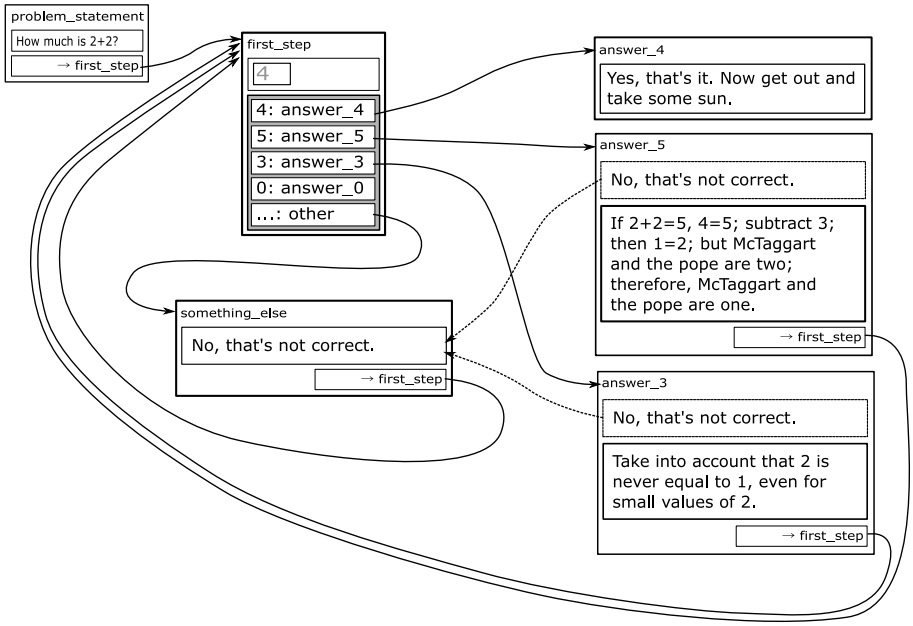
The exercise representation was done by George Goguadze and the author based on a paper by Goguadze et al.[Gog03], and refined later by converting the tree into a directed graph in collaboration with Manolis Mavrikis. Further improvements were done by George Goguadze and the author.

Exercises are directed graphs, whose nodes are called "interactions". An exercise step consists of one or more interaction nodes.



Interactions can be complete, or degenerated. Complete interaction nodes have an answer map, and require input from the user. Degenerated interaction nodes do not include an anwer map, and are used as containers for feedback reuse. For instance, a common occurrence is that there are a variety of wrong answers to a specific problem, but each of them displays a different misconception in the mind of the user, so that the feedback for each includes a general indication of the answer being wrong (which can be textual or graphical: e.g. decorating the given answer with different colors) and a specific one meant to correct the misconception. This corrective feedback can contain any static content allowed in ActiveMath (including links to course materials), and also sub-exercises that when finished will let the user continue with the original one.

These features can be seen in the following example, where "No, that's not correct" is the generic feedback, which is added to the more specific ones given for concrete answers:



Neither the system nor the exercise format make any distinction between exercises and sub-exercises: both are treated just as graphs, with sub-exercises being sub-graphs of the main one.

Branches occur at the answer maps. It maps the user's answer to different feedbacks, depending on a classification method. The implementation in ActiveMath uses equality conditions as the classifier, but other methods such as bayesian filters are possible.

The details of the knowledge representation and how it works in the ActiveMath exercise subsystem architecture are published in [faia05].

3 Design of the exercise system

The exercise system consists of a kernel, that handles exercise content in OMDoc and user's answers given in OpenMath, and a user interface mediator, that formats the OMDoc content for display (into HTML, etc.) and translates the user's answers into OpenMath.

The output formatting is done by the presentation system in ActiveMath, using mainly XSLT stylesheets, and is not discussed here.

The input translation is done in different ways depending on the input method used. The default one is an HTML form, whose content gets parsed into a list of OpenMath expressions, according to a selectable grammar. As of this writting, there are 8 grammars available, corresponding to the syntaxes of several mainstream Computer Algebra Systems. It is also possible to give the answer directly in OpenMath.

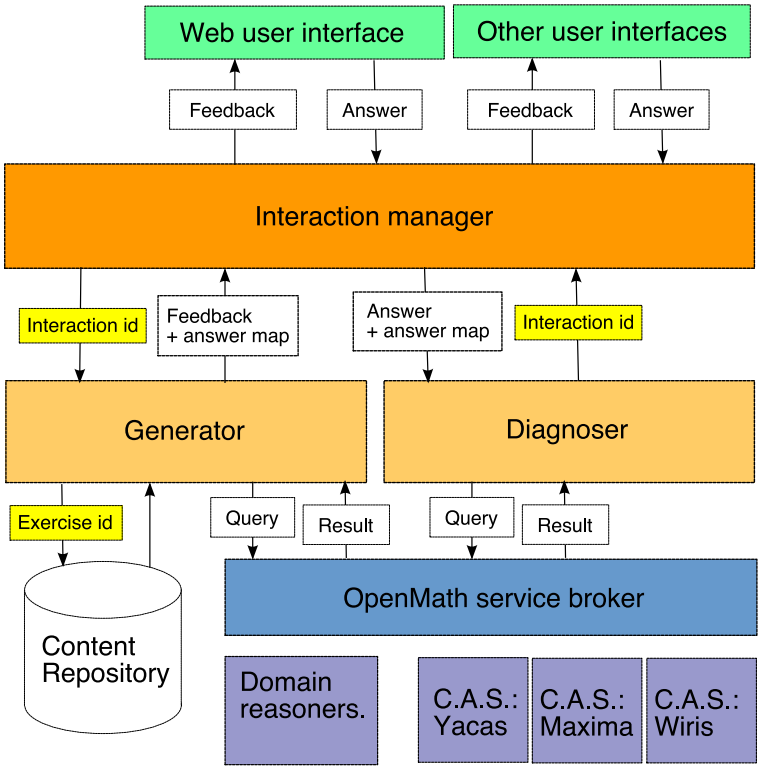
Other input methods, such as figure drawing, are possible as long as the input is translated in some way to OpenMath expressions. For instance, an interface for geometry exercises would translate the geometrical loci (not the drawn lines) into mathematical expresions like "circle(10,4,3)".

For the classification of the answers, there are several evaluation services that provide formula evaluation of different kinds:

- Numerical approximation.
- Symbolic simplification (canonicalization).
- Rule application step by step (domain reasoning).

The evaluation service to use is picked by the OpenMath service broker according to the kind of evaluation needed, and the requested evaluation context, that specifies which mathematical symbols should be evaluated. As a minimum, the broker needs to find a service that knows about those symbols, which depends on the OpenMath phrasebook[10].

[10] This is a translator between OpenMath expressions and some external service such as a CAS.



The objects exchanged are as follows:

Feedback: JDOM Element object tree, top element <feedback>. This includes all the content displayed to the student, including all the feedback to his actions: the problem statement is the feedback to the action of starting the exercise, there is feedback for each answer, and for requests such as "hint", "solution", etc.

Answer_map: JDOM Element object tree, top element <answer_map>. The mapping between conditions and diagnosis identifiers (Interaction_id): the evaluator uses it to classify the answer in one of the given categories, and gives back the identifier assigned to that category to the interaction manager.

Exercise_id: String. This is just the unique identifier for the exercise in ActiveMath's content repository.

Interaction_id: String. A unique identifier for an interaction node, which is used to request the next interaction content from the generator.

Answer: JDOM Element object tree, top element <OMOBJ>. This can actually be any kind of XML structure that the Evaluator can handle, and currently it is always OpenMath expressions.

3.1 On-demand graph generation

The interaction manager only sees one node of the graph at a time.

The nodes for the graph are generated on demand by the exercise generator in use. The interaction manager asks for a specific node by its unique identifier.

The node generator produces a whole node when receiving an identifier, and puts the possible next nodes into the transition map, together with the conditions associated with each transition.

The evaluator/diagnoser finds the condition that matches the input, and gives back the identifier for the node that contains the feedback for that condition, which is then given to the generator for the next step.

The "static" generator loads a pre-determined exercise file (graph), and extracts nodes from it on demand from the interaction manager.

There is an inheritance hierarchy of generators, that allow to specialize an existing one. For example, the "randomizer" generator extends the "static" one so that, after loading the static content, it gets modified by replacing the specified variables by a randomly-chosen expression.

Generators can be written in Java or, by using the "XSLT" generator, in the XML Stylesheet Language defined by the W3C[XSLT]. In this last case, the XSLT is included in the exercise content, to ensure that it will not be inadvertedly modified by some other author and render the exercise unusable.

3.1.1 Using generators

An exercise can be built using a generator by using the "interaction_generator" element with the generator name given as an attribute. In this case, the exercise composer first loads the exercise content, and then applies the requested generator to it, passing the parameters given as "parameter" subelements.

For instance, to apply the randomizer generator to an exercise, the following can be added directly inside the "exercise" tag:

```
<interaction_generator name="Randomizer">
  <parameter name="constant">
    <OMOBJ><OMV name="n"/></OMOBJ>
    <OMOBJ><OMI>1</OMI></OMOBJ>
    <OMOBJ><OMS cd="nums1" name="pi"/></OMOBJ>
    <OMOBJ><OMA><OMS cd="arith1" name="plus"/>
                <OMI>7</OMI>
                <OMI>4</OMI>
           </OMA>
    </OMOBJ>
  </parameter>
</interaction_generator>
```

The randomizer will then pick either 1, π, or $7 + 4$, and replace all occurrences of n by it. Please note that the last expression is not evaluated, so $1 + n$ in the exercise content would be replaced by $1 + 7 + 4$, not by $1 + 11$ or 12. If

evaluation of the replacement expression is desired, it can be requested by using the parameter name "evaluated_constant" instead of "constant":

```
<interaction_generator name="Randomizer">
  <parameter name="evaluated_constant">
    <OMOBJ><OMV name="n"/></OMOBJ>
    <OMOBJ><OMA><OMS cd="arith1" name="plus"/>
                <OMI>7</OMI>
                <OMI>4</OMI>
           </OMA>
    </OMOBJ>
    <OMOBJ><OMA><OMS cd="arith1" name="times"/>
                <OMI>7</OMI>
                <OMI>4</OMI>
           </OMA>
    </OMOBJ>
  </parameter>
</interaction_generator>
```

In this case, n would be replaced by either 11 or 28, so the previous example, $1 + n$, would be replaced by either $1 + 11$ or $1 + 28$.

The evaluation is done by the same Computer Algebra System used for answer classification.

Each defined constant can be used in subsequent constants: for instance, a constant "m" defined after "n" as $n + 1$ would have the randomized value of "n" taken into account, as shown in the following table for $n = 1+4$ and $m = n-3$:

parameter names for n and m	value of n	value of m
constant, constant	1+4	1+4-3
evaluated_constant, constant	5	5-3
constant, evaluated_constant	1+4	2
evaluated_constant, evaluated_constant	5	2

3.2 User input

The exercise system kernel works only with OpenMath expressions, so there has to be some kind of user interface. The author implemented a set of parsers that allow the student to type his answers in a choice of linear syntaxes from several Computer Algebra Systems.

The parsers produce the OpenMath equivalent of the input expression, which is then sent by the user interface to the interaction manager, that in turn gives it together with the transition map to the answer classifier for feedback mapping.

The grammars are just small subsets of the complete languages of these systems, as we need only to parse the kind of expressions that are given as answers by the students in our exercises. The purpose is to allow students familiar with any of those languages to start using ActiveMath immediately without having

to adapt to yet another syntax. The equivalences between different grammars were taked from the Rosetta Stone for Computer Algebra Systems[Rosetta].

Java applets, such as formula and graphic editors, can give their results in any of those formats (by specifying which one), although in practice it is done in OpenMath.

3.2.1 Parser details

The syntaxes included with ActiveMath are the following:

Syntax	$\sin 2x^3$	Notes:
Yacas	Sin(2*x^3)	
Maxima	sin(2*x^3)	Case-insensitive
Axiom	sin(2*x**3)	Case-insensitive
Mpl (Maple™)	sin(2*x^3)	
Mma (Mathematica®)	Sin[2x^3]	
Mpd (MuPAD)	sin(2*x^3)	
Drv (Derive)	sin(2x^3)	Case-insensitive
Rdc (REDUCE)	sin(2*x^3)	Case-insensitive

The OpenMath form of the above expression is as follows:

```
<om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
  <om:OMA><om:OMS cd="transc1" name="sin"/>
          <om:OMA><om:OMS cd="arith1" name="times"/>
                  <om:OMI>2</om:OMI>
                  <om:OMA><om:OMS cd="arith1" name="power"/>
                          <om:OMV name="x"/>
                          <om:OMI>3</om:OMI>
                  </om:OMA>
          </om:OMA>
  </om:OMA>
</om:OMOBJ>
```

The "foreign" function allows to construct valid expressions from subexpressions that use different syntaxes. It takes a syntax identifier as first parameter, and an expression as second. For instance, to parse the Mathematica®expression "Sin[x]" from within the Maple™-style parser:

```
Mpl> foreign("Mma", "Sin[x]")
<om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
  <om:OMA><om:OMS cd="transc1" name="sin"/>
          <om:OMV name="x"/>
  </om:OMA>
</om:OMOBJ>
```

The "foreign" function works at any place in an expression:

```
Mpl> sin(x) + foreign(“Mma”, “Sin[x]”)
<om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
  <om:OMA><om:OMS cd="arith1" name="plus"/>
          <om:OMA><om:OMS cd="transc1" name="sin"/>
                  <om:OMV name="x"/>
          </om:OMA>
          <om:OMA><om:OMS cd="transc1" name="sin"/>
                  <om:OMV name="x"/>
          </om:OMA>
  </om:OMA>
</om:OMOBJ>
```

The exact syntax of the “foreign” function follows the style and grammar of each parser:

```
Mma> Sin[x] + Foreign["Mpl", "sin(x)"]
<om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
  <om:OMA><om:OMS cd="arith1" name="plus"/>
          <om:OMA><om:OMS cd="transc1" name="sin"/>
                  <om:OMV name="x"/>
          </om:OMA>
          <om:OMA><om:OMS cd="transc1" name="sin"/>
                  <om:OMV name="x"/>
          </om:OMA>
  </om:OMA>
</om:OMOBJ>
```

Finally, it also works with the syntax identifier “OpenMath”:

```
Maxima> x*foreign("OpenMath", "<OMOBJ><OMV name='y'/></OMOBJ>")
<om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
  <om:OMA><om:OMS cd="arith1" name="times"/>
          <om:OMV name="x"/>
          <om:OMV name="y"/>
  </om:OMA>
</om:OMOBJ>
```

3.3 Expression evaluation

The “OpenMath broker” provides access to an appropriate OpenMath service for the given parameters, the computation type and the computation context:

```
evaluator = OpenMathServiceBroker.getService(type, context);
```

First is the computation type, which is one of the following:

NUMERICAL_IEEE754_DOUBLE_FLOAT: the expression is to be evaluated using IEEE-754 double precision floating point numbers, and the final result expressed as an OpenMath floating point object, OMF.

SYMBOLIC_BASIC_SIMPLIFY: the expression will be evaluated by some symbolic processing package (typically a Computer Algebra System), using some basic simplify method. The results are not guaranteed to have any specific form, since the meaning of "basic simplification" varies from program to program. This is typically used for comparing two expressions, by subtracting them and simplifying the result, which will typically produce "0" if they are algebraically equivalent according to the rules defined in the symbolic processor.

LOGICAL_BOOLEAN: the expression is supposed to contain only symbols from the "logic1" OpenMath CD, and the final result is either the OM symbol "logic1:true" or the symbol"logic1:false".

The second parameter is a "context" for the computation, specifying which symbols will have to be processed by the provided OpenMath service. For example, the simplification function used in the CAS depends on the expression: in Yacas, for expressions involving polynomials the function used is Simplify(), while for expressions involving trigonometric functions the appropriate simplification function is TrigSimp(). [not implemented yet]

3.3.1 Expression evaluation in a specific context

When checking whether an answer given in an exercise is correct or not, we want to accept as valid not only a specific answer, but also those variants that do not affect the abilities being tested. For instance, when asking the student to find the derivative of an expression we want to accept answers that are algebraically equivalent to the canonical (as defined by the exercise author) answer, such as $\frac{x}{2}+1$ instead of $\frac{x+2}{2}$. The way we do it is to canonicalize the answer by evaluating it in a specific context in which the algebraic simplification rules and arithmetic operations for "plus", "minus", "divide" and "times" are defined, but not the ones that the student should perform such as the derivation rules. This is not the usual filtering: we do not reject a priori answers that contain such operations but pass them to the system, to allow the exercise author to provide specific feedback to the student in those cases and to let those answers get reported to the user model, as this could indicate some specific misconception in the student, such as mistaking the derivation exercise for a CAS usage exercise in which his task would be to translate the problem statement into a CAS expression that computed the answer.

3.3.2 Communication with external systems

There are four mechanisms currently implemented for connecting to external systems such as CAS and domain reasoners:

1. Inter-process communication using system pipes: the external system is started from ActiveMath as a sub-process. Then, the expression to be evaluated is sent to the sub-process standard input, and the output and errors are collected using the sub-process standard output and error channels. The communication with the CAS Yacas is done in this way.

2. XML-RPC (Remote Procedure Call): the system is assumed to be already running as an XML-RPC server. The expression to be evaluated is given as a string parameter to the procedure "OMEval.evaluate", and the returned string is parsed as XML. This is how we connect to Wiris in the frame of the LeActiveMath project.

3. WSDL (Web Service Description Language): an already running WSDL server is sent the expression to evaluate, and the result parsed back as XML. The connection to the CAS Maxima works in this way.

4. Interprolog[11] library for communication between Java and Prolog: the expression to be evaluated is translated to an appropriate form for the given Prolog domain reasoner, and the result Prolog term tree is translated back to OpenMath.

The broker chooses the adequate method depending on the value of the "exercises.cas.command" property:

- If it begins with "http://" and ends with ".wsdl", the WSDL method is used.
- If it begins with "http://" and is not a WSDL file, we connect to it using XML-RPC.
- Otherwise, the value is supposed to be the command line for starting an external process, using the first method listed above.

3.4 CAS console

The availability of the input parsers and the OpenMath CAS connection allows the use of the configured CAS using any of the input syntaxes, independently of the CAS used for evaluating the expressions.

The CAS console makes use of all the resources of the exercise system to provide its functionality with a minimum of program code. It only needs to take the given input from the exercise user interface, and evaluate it in the CAS via the regular connection (the OpenMath broker) to get the result to be displayed to the user, using the feedback display mechanism of the exercise system.

It has been implemented as a "virtual exercise" (section 3.1), with the CAS evaluation of the input being done in the custom interaction generator.

[11] http://www.declarativa.com/interprolog/

It is also possible to access this functionality from the command line interface. The command to launch it is "bin/amcas" for POSIX systems, and "bin\wamcas" for Windows. It works as follows:

```
#!/bin/sh
java -classpath lib/activemath.jar:lib/thirdparty.jar
    org.activemath.exercises.openmath.OpenMathServiceBroker
```

4 Domain reasoning and diagnosis

The domain reasoning queries to the OpenMath broker provide, depending on the functionality of the specific domain reasoner, solution graphs that can be used for producing exercises and for diagnosis of the user's answers, or directly the feedback for the user after doing the diagnosis itself. In any case, the interface to the exercise system are the InteractionGenerator and Evaluator classes.

A domain reasoner applies the rules available to it (for a specific domain) until some canonical form of the expression is reached (the "solution"), and outputs the "solution graph" that contains each rule application and the partial results at each step in the processing. There are two domain reasoners being integrated into ActiveMath: one for Calculus written in Prolog by Claus Zinn, another for Statistics written also in Prolog by Barbara Grabowski.

The output of the domain reasoner can be used in diagnosis to find where in the solution path the student is, so that proper feedback is given. This area will be developed by George Goguadze.

5 Implementation details

5.1 Kernel

The kernel is the self-contained part that runs exercises independently of the content storage (database, files, etc.) and user interface, as all its input and output is JDOM objects that model OMDoc trees or OpenMath expressions.

5.1.1 Node generator factory (exercise composer)

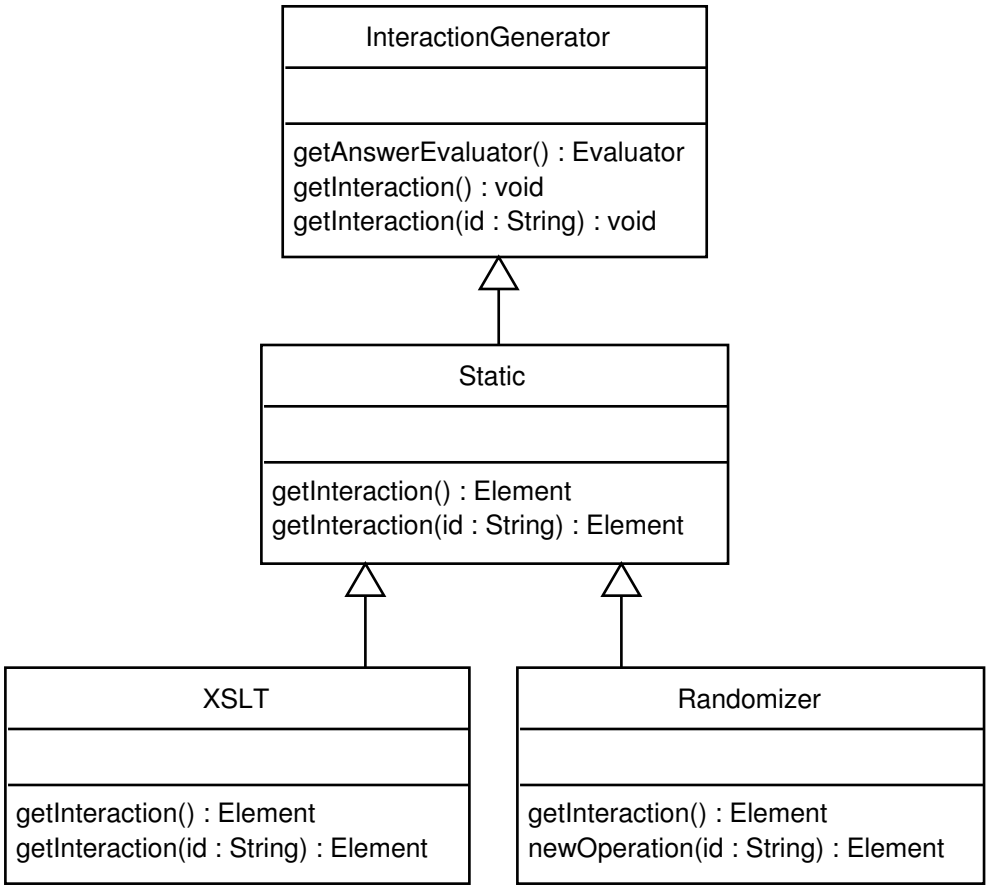
This factory is asked to produce a generator object for a given exercise as soon as the user requests it.

5.1.2 Generator

The generator class has the responsability of producing the feedback to be given to the user after each action, which can be of the following kinds:

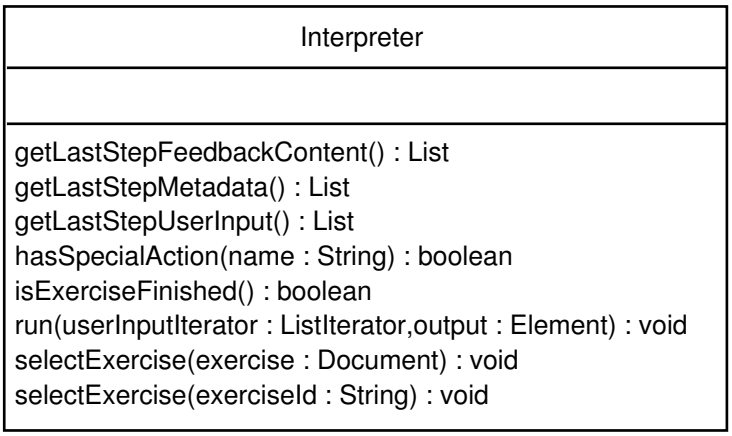
- Starting the exercise: in this case, the feedback is the problem statement.

- Giving an answer: this is encoded as an OpenMath expression in our current implementation, but could be any other XML tree that the Evaluator could match to the given answers in the answer map.
- Requesting help: this includes asking for hints and giving up.



5.1.3 Interaction manager

The exercise interpreter orchestrates the interaction process by requesting the generator for the feedback and answer maps in each step, sending the feedback to the user via the user interface mediator, getting the user's answers from it and passing them along with the answer maps to the evaluator, and finally getting the diagnosis from the evaluator and giving it to the generator so that it can construct the appropriate feedback.



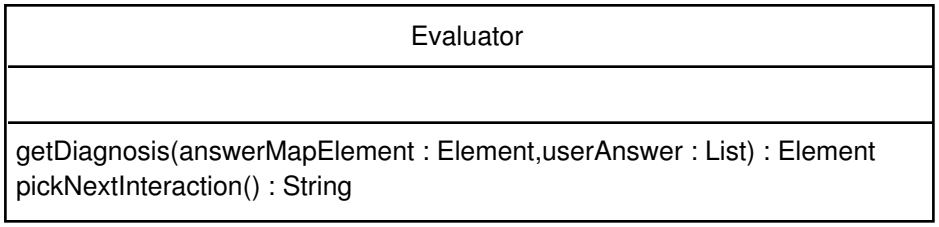
5.1.4 Evaluator

The evaluator classifies the user answer in one of the categories given in the answer map, which is the diagnosis process. The current implementation provides three kinds of comparisons of the user input with a given mathematical expression:

1. Numerical equivalence: the user's answer is matched if it is a number (integer, floating point, or rational) and the difference, when evaluated as a floating point number, is less than a given epsilon.

2. Syntactical equivalence: the expression trees are compared literally, with very little normalization: OpenMath error objects are compared only up to their head symbol, ignoring any extra information such as string messages. The reason is to be able to match error kinds regardless of whatever explanations are given by remote systems accessed through the OpenMath service broker.

3. Sematical equivalence: this uses some external system to check whether the user's answer is semantically equivalent, that is, has the same meaning according the external system, as the entries in the answer map. That external system is typically a Computer Algebra System, although it can be also a domain reasoner or any other service capable of comparing mathematical expressions.



5.2 User interface mediator

The user interface mediator is responsible for translating the feedback given by the exercise system as OMDoc JDOM objects into something the user can read, such as HTML or PDF, and translate the user's answer into OpenMath expressions understandable by the kernel.

5.2.1 Webapp exercise controller

The default way of interacting with the system is an HTML form that contains the two basic interactive elements, blanks and selection lists, with different styles each.

The form includes a selector for the input syntax, that specifies how to parse the answers given in the blanks. Additionally, the system adds buttons for "user requests", such as asking for hints or giving up the exercise.

When a page includes both normal blanks and special ones, such as applets, it's necessary to allow the special input elements to give their values in a way independent of the normal blanks for which the user selects the syntax. This is accomplished using the "foreign" function available in all the parsers, as described in section 3.2.1 "Parser details", page 9.

The interactive elements are adressed by their index. This has the problem that the shuffling done by the presentation systems alters it for some symbols, such as the summatory, for which the MathML representation requires some elements to be in a different order from the OpenMath representation. When replacing them by interactive elements, the final result would produce an expression list with the wrong order, since the conditions in the answer map use the original positions.

An easy solution for this would be to use unique identifiers for each interactive element, and use it always to refer to it. The autor decided not to do it this way because choosing identifiers and referring them is a tedious and error-prone process. Instead, the system assigns identifiers to the elements just before sending the document fragment to the presentation system, and uses that information to recover the original ordering in the user interface part, which reorders the answers before sending them back to the interaction manager in the intended order. This way the whole process is transparent to both exercise authors and users.

5.2.2 HTML form and Javascript functions

We allow an arbitrary number of interactive elements on the page. To give back a single combined answer, when the user submits the form all inputs are collected into a single valid mathematical expression (that can contain string objects) and sent to the exercise controller described before.

5.2.3 Decoration of previous answers for feedback

The user's answers can be decorated according to the diagnosis from the evaluator. There are three predefined values in ActiveMath:

good the answer was a good one, that is, both correct and relevant for the task at hand.

bad the answer was a bad one, which normally indicates some misconception by the user.

near the answer does not qualify as "good", but it is going in the right direction.

Other values are possible, and in the test exercises we defined two more for demonstration purposes: "crying" and "dancing", which decorated the user's answers with cartoons performing those actions in addition to the usual green/red/yellow coloring.

References

[Gog03] George Goguadze et al.

[Rosetta] The Rosetta Stone for Computer Algebra Systems: `http://www.univ-orleans.fr/EXT/ASTEX/astex/doc/en/rosetta/htmla/roseta.htm`

[XSLT] XSLT version 1.0

[faia05] Interactivity of Exercises in ActiveMath, G. Goguadze and A. González Palomo and E. Melis, in "Towards Sustainable and Scalable Educational Innovations Informed by the Learning Sciences Sharing. Good Practices of Research Experimentation and Innovation", C.K. Looi and D. Jonassen and M. Ikeda, editors, ISBN 1-58603-573-8.