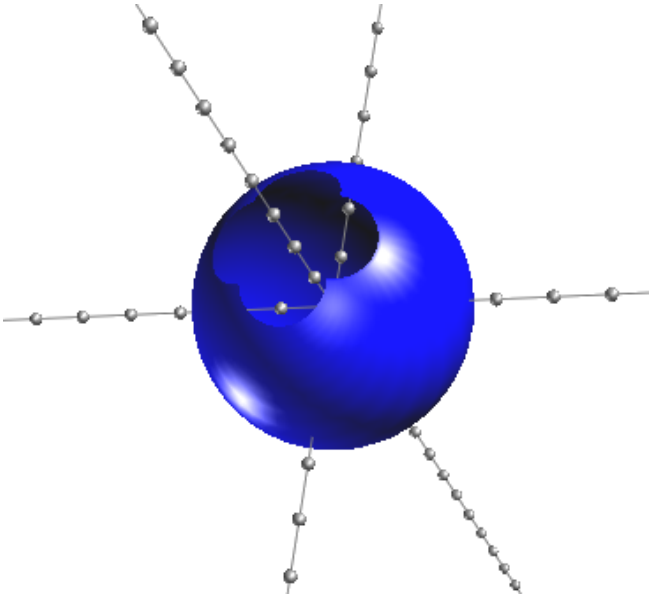
The Mumie
MathletFactory

An Open Source Java Library for Interactive
Mathematical Applets (*"Mathlets"*)



Author:
Markus Gronau

February 18, 2015

The picture on the front page was taken from a screenshot of the mathlet `SurfaceInR3` and shows a parameterized surface in $\mathbb{R}^3$.

Contents

1 Abstract **3**
1.1 About this Document 3
1.2 About the Mumie MathletFactory 3
1.3 Fundamental Concepts 4

2 Applet Skeleton Framework **5**
2.1 Mathlet Instance Methods 5
2.2 User Interface 6
2.3 Multilinguality 7
2.4 Help System 8
2.5 Error Handling 8
2.6 Common appearance 8

3 Model-View-Controller Architecture **9**
3.1 Model Architecture 9
3.2 View Architecture 10
3.3 Controller Architecture 11
3.4 Architecture Overview 12

4 Mathematical Models **14**
4.1 Arithmetic Model 14
4.2 Algebraic Model 15
4.3 Geometric Model 16
4.4 Vector Space Model 16

5 Homework Architecture **18**
5.1 System Overview 18
5.2 Homework Receipts 19

6 Outlook **20**

1 Abstract

1.1 About this Document

This document reports on the MUMIE MATHLETFACTORY (versions 2.0 and 2.1), an open-source Java library for interactive mathematical applets. This document gives a structural overview of the main features but does not intend to describe in detail the neccessarities for building own programs. These information are rather contained in the MATHLETFACTORY *tutorial*.
Some of the mentioned features are only available in the milestone release 2.1.

1.2 About the Mumie MathletFactory

The MUMIE is an e-learning platform specialized in mathematics and mathematical sciences. It is a fully web-based learning- and teaching environment using standard techniques (such as XHTML, MathML and Java) for deploying mathematical content to the end-user inside a standard compliant internet browser.

The MATHLETFACTORY is a Java library and part of the MUMIE in which it is used to produce and to visualize dynamic, interactive mathematical content with Java applets (so called "mathlets"). It allows the rapid development of such mathlets, containing complex mathematical algorithms and scenes along with a common generic behaviour and appearance.
The MATHLETFACTORY library contains a large collection of mathematical objects (so called MM-OBJECTS), which can be used both in calculations and presentations of dynamic problems. Their visualizations may be both symbolic and graphical (both 2D and 3D). Their state may be interactively changeable by the user and may cause further interaction between them.

The MATHLETFACTORY is developed since 2001 and released in 2007 the milestone 2.0, which has been used since then by several thousands of students at different international universities. The main development effort is actually done for the milestone 2.1 which will be released in spring 2008.

The MATHLETFACTORY library is open source and provided under the MIT license. It is compliant with all Java versions of SUN Microsystems and Apple Inc. starting with Version 1.4.2[1]. Further information and documentation as well as source code is available under `http://www.mumie.net`.

[1] Some additional extensions may require a newer Java version.

1.3 Fundamental Concepts

Generic programming of mathlets

The programming of mathlets is generic and easy to use but does not restrict the applet developer in his creativity. By automatically adding standard features to new mathlets and providing a flexible and generic applet "skeleton" the developer can concentrate on the mathematical content.
The MathletFactory library acts hereby as a reusable component system.

Separation of logic and representation

The MathletFactory follows the philosophy to separate the (abstract) mathematical object from its further representation(s) on the screen, defining a veritable Model-View-Controller architecture (MVC). By handling interactive actions (e.g. user interaction) through generic but specific *events*, changes are automatically reflected to the mathematical model and also propagated to any dependant objects, allowing even complex dependency trees.

Abstract number fields for arbitrary-precise calculations

Most mathematical objects are based on an abstract number class which makes it possible to perform calculations on a particular number field with its own arbitrary precision. While some operations need a complex number field, other situations can be more satisfied with an answer in whole numbers. Furthermore while e.g. floating point operations are executed faster than rational ones, the latter are more precise and user friendly.

Open extensible framework

The MathletFactory library offers a wide spectrum of objects for the most needed mathematical entities and applet developer's concerns but also can be extended in almost every included technology[2]. By providing an open framework and both a complete API documentation and tutorials for its techniques, the MathletFactory truly underlines its open source idea.

[2]Some extension features are only available in the up-coming milestone 2.1

2 Applet Skeleton Framework

With providing a flexible and powerful "skeleton" framework for applets, the developer has neither to "reinvent the wheel again", nor to transfer whole passages of code for new programs. Instead, basing on an applet skeleton, mathlets have a generic interface for both the code itself and the placement of code. This simplifies the understanding of mathlet code (even if it is not the "own" code) and thus reduces developing and debugging time.

2.1 Mathlet Instance Methods

Observing the life cycle of Java applets, these offer several *entry points* for actions to take place according to the current applet's *state*. One of these is the `init()` method which will be called after the applet class is initialized. Observe that many applet specific features are not available till the `init()` method of the applet is reached. Calling this method is necessary for properly initializing the mathlet's runtime, subclassing applets must be aware of this. Since every mathlet extends one of the skeleton classes, every feature can be easily accessed via one of the mathlet's instance methods but only after the runtime's initialization.

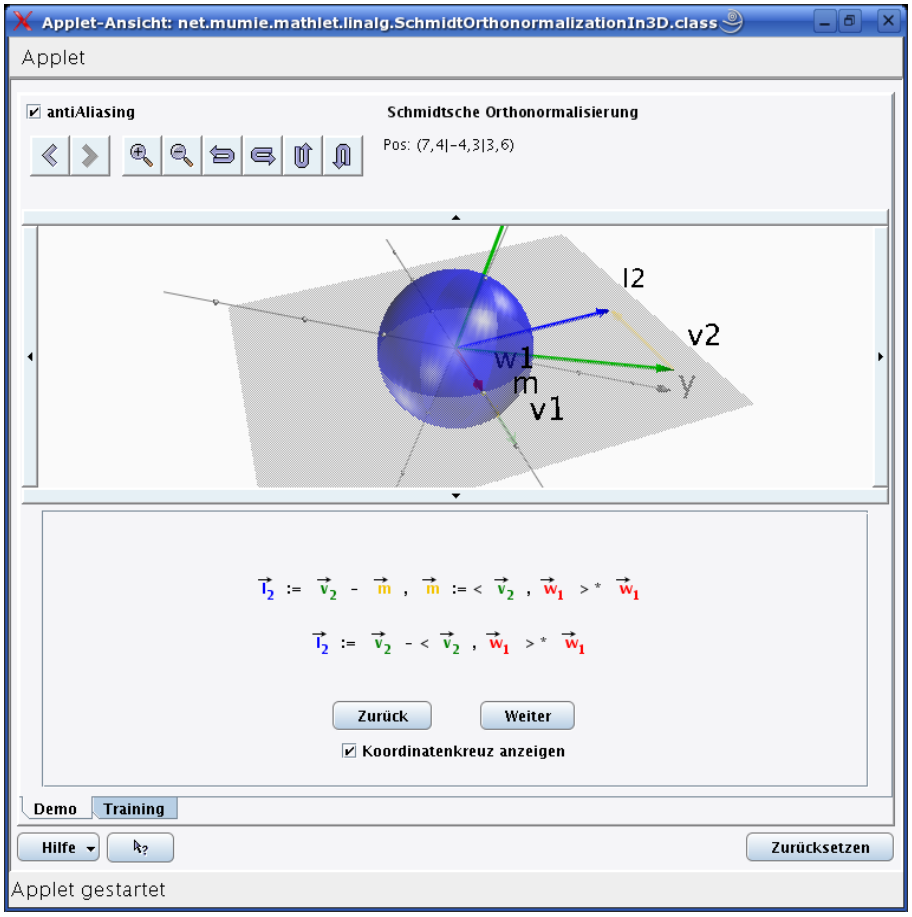


Figure 1: Screenshot: Schmidt Orthonormalization

2.2 User Interface

Creating user interfaces in Java can be a very complex task: on the one hand code can become very large for even simple interfaces, thus making it hard to maintain it. On the other hand a common appearance with a distinctive design across all programs is wanted. These implications were resolved in two ways: first by providing basic layout templates which can be used as a fundament for new mathlets and secondly with powerful layout techniques which allow to arrange components in a "text editor like" manner and thus permitting a "human readable" and very short layouting code without a long training period into Java SWING or the MATHLETFACTORY library.

Skeleton Templates

Every mathlet's layout defines three basic parts: the title, the content and (bottom) button area. The different skeleton classes only differ in the content area, which may include one or more graphical drawing boards (each a 2D or 3D CANVAS) or none. All of them include the functionality of an integrated `ControlPanel` to display symbolic and descriptive content underneath the canvas/canvi.
The fundamental skeleton class of all further template classes is `BaseApplet` and may also be used as a basis (i.e. super class) for own applets.

Laying out Control Elements and Text

Each mathlet defines a single `ControlPanel` but may also use several of them, even nested into each other. Content in a `ControlPanel` is arranged in a text editor like manner i.e. into rows. Each row alignes its content from the left to the right and can have its own horizontal alignment. Inserting a *line break* causes the current line to end and to start a new line which will be used for the further adding of components: `insertLineBreak()`.
Adding text can be achieved via `addText(String text)` or with one of the `addText` methods requesting an additional color and/or font argument.

Entering Descriptive Content

Descriptive texts beside mathematical formulas are essential inside an educational program like mathlets. For this purpose, three common text "formats" are available: TeX, HTML and plain text. Each format can be used in any text component and offers different advantages compared to the other two.

Common Baseline Sharing

As the bridge between textual and symbolic content, every UI component supports a common baseline sharing with all other elements aligned in the same row (e.g. inside a `ControlPanel` or a text viewer). This technique allows an ordered and clearly arranged layout of e.g. mathematical formulas surrounded by descriptive text blocks.

2.3 Multilinguality

Every language sensitive component inside the MATHLETFACTORY library uses the MESSAGES technology which allows to combine an abstract *message key* with a concrete translation (the real *message*) during execution time. By this way no programming code must contain any language dependant expressions, thus permitting to maintain any descriptive content beside the source code and to add translations for new languages for already existing features.
New messages can be defined as stated in the *Java Property Format* specification in a `key = value` notation in a file named `Messages_[LANG].properties` where [LANG] means the language's code. They can be retrieved inside a mathlet with the instance method `getMessage(String key)`.

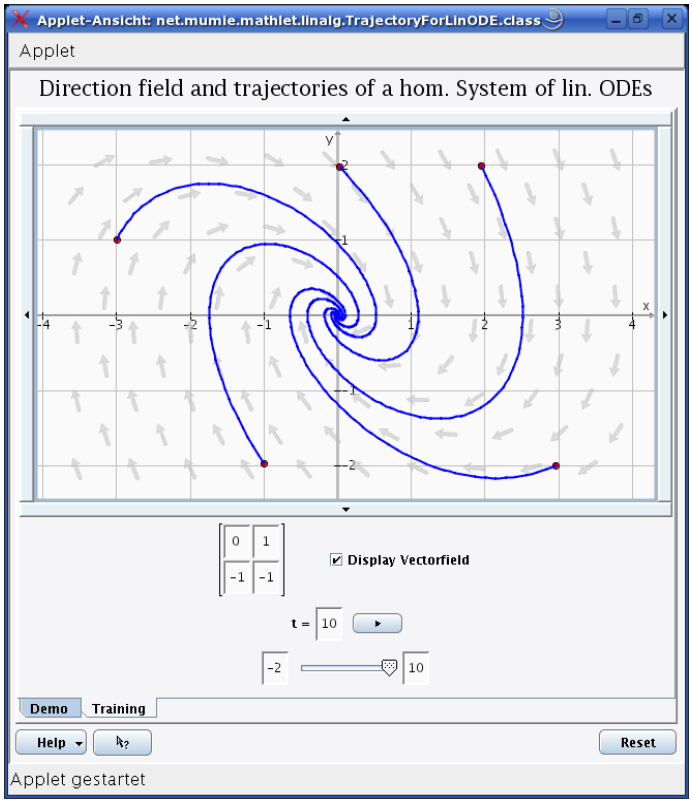


Figure 2: Screenshot: Trajectories for a Linear Ordinary Equation System

2.4 Help System

The MathletFactory supports two types of help systems: if the JavaHelp extension is available, the (default) help system can be used providing various standard features like HTML and print support, a navigatable table of contents or a context sensitive help. If the default help system cannot be used for any reasons a simpler built-in system will be loaded instead, permitting a less confortable but sufficient help.
Like all language sensitive technologies, both help systems respect the mathlet's language setting and may be extended for new languages but also with additional help topics.

2.5 Error Handling

Due to the complexity of software systems and their configurations (including the internet browser and the *Java Plug-In*) a flawless program execution cannot be assured, even if the system can be declared as *stable*. These *runtime errors* may be caused by unpredicted user actions, defective network connections, faulty configured systems or simply software bugs and should be catched and processed by an appropriate error handling system, which notifies the user and proposes useful solutions for the problem. While Java applets always run on the client side, an error report describing the failure and including system information may be useful for further debugging and bug fixing procedures of the administrating staff.
For this purpose the MathletFactory includes a complete quality feedback system with automatical generating error reports, which can be viewed and probably sent to a Mumie server by the user.
At this point, based on the experiences with error reports from students, the MathletFactory is able to use some workarounds and bug fixes for various incompatibilities between the multiple Java versions and vendors.

2.6 Common appearance

Per default every mathlet uses an elaborated theme (the `MumieTheme`) for the *Java Metal Look&Feel* which assures a common appearance across all operating systems and Java versions/vendors and updates the default Java theme with a clearer and more modern look. This theme allows a scalable appearance for all mathlets including color, font and other *look and feel* relavant settings.

3 Model-View-Controller Architecture

The MATHLETFACTORY holds on the paradigm to separate the logic from its representation on the screen, thus introducing various *interfaces* between the single technologies. These interfaces guarantee an implementation of new features which is independant of restrictions of already exisiting components. Hence this "Model-View-Controller" architecture simplifies the creation of new mathematical objects and allows a clear distinction between the mathematical *model*, the *view* on the screen and the *controller* processing user actions and updating the model. See figure 3 for an overview of the components in such an architecture.

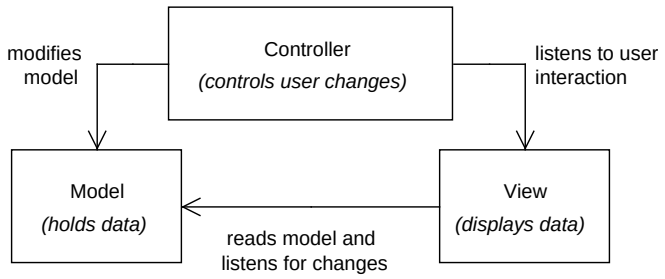


Figure 3: Common role allocation in a MVC architecture

3.1 Model Architecture

In the MATHLETFACTORY there are two *grades* of models: first the pure mathematical model describing the internal state of a mathematical entity/object (the M-OBJECT) and secondly the extending *multimedial* model containing additional *display properties* and interfaces for the MVC architecture (the MM-OBJECT).

Mathematical Models

A mathematical model or a M-OBJECT holds the internal data of an entity and offers methods for operating with it. M-OBJECTS are completely detached of any viewing and interaction capabilities, thus forming a *calculation layer*.
They are usually based on an abstract NUMBER CLASS which is used to interchange number values with other M-OBJECTS and to execute primitive calculations (see section 4.1).
The M-OBJECT interface was deliberately designed as an *open interface* thus a M-OBJECT is free of any restrictions.

MM-Objects

MM-Objects are both mathematical and multimedial components and are the primary entities with which a developer has to deal with. They typically extend a M-Object and add the MM-Object capabilities via interfaces[3], thus forming an *interactivity layer*.
Some MM-Objects incorporate their mathematical model for simplicity.

3.2 View Architecture

The view of a MM-Object is its representation drawn on the screen which is performed by a *drawable*. The drawable's data is transformed from the mathematical model to a display model via a *transformer*. Using an abstraction layer between these two models allows different abstract presentations of the same mathematical content.

Transformers

A transformer is responsible for the correct adjustment of drawables depending on the current MM-Objects's state. Every time a MM-Object needs to be rendered, the transformer's `render()` method will be called, allowing to update the display model and the view(s).
A transformers is qualified by the master's type (i.e. the MM-Objects's type), a screen type (e.g. 2D, 3D, "no canvas" for symbolic views) and a transform type, which describes the transform into the display model. Every MM-Object defines a default transform type for both symbolic and graphical representations. The base class of all transformers is `GeneralTransformer` which is also responsible for registering the transformer mappings and creating the appropriate transformer instances on demand.

Drawables

"Drawable" is the abstract name of a viewing component, which itself can composed by various *canvas drawables* for graphical views or by *container drawables* for symbolic views. Drawables are created and updated by transformers and are designed to be reusable in new representations.

[3] due to the fact that multiple inheritance is not possible in Java

3.3 Controller Architecture

The controller architecture differentiates between actions in a canvas and those in a SWING container. The later use a different, easier technique since container drawables can only be edited on one way: with the keyboard. A canvas instead, when encountering an user action, propagates it to its *canvas controller* who generates a corresponding `MMEvent` and notifies the appropriate handler dealing with this action. These modify the MM-Object and invoke an update of dependant objects, which is done by *updaters*. Actions can originate from the mouse and/or the keyboard.

Handlers

Handlers track mouse or keyboard actions (or a combination of both) in canvas drawables and react accordingly. Each handler is characterized by a `MMEvent` which describes an action the handler wants to deal with. There are two different types of handlers: object specific and global handlers. The first are added to a MM-Object and modify its internal state while the later are added directly to a canvas and transform the view of a scene.
The MathletFactory contains a large number of predefined object specific handlers for the personal use.

Updaters

Updaters track changes in dependant MM-Objects and adjust the *slave* MM-Object they were added to accordingly. As the simplest implementation, the `DependencyUpdater` allows to adjust MM-Objects by placing arbitrary code on-the-fly. Besides there is a large number of predefined updaters for the most common needs.

Events

Events (or more precisely `PropertyChangeEvents`) are used to inform the master MM-Object about changes in symbolic drawables. This is typically due to user interaction inside the SWING container where a content string, describing the drawable's data, can be edited in a textfield. If the new input is validated, an event is fired to the master MM-Object which adjusts its internal state and propagates the changes.

3.4 Architecture Overview

The main components of the MVC architecture as well as their primary functions are:

MM-Object	both mathematical and multimedial component
Transformer	transforms mathematical model and updates drawable's view
Drawable	visualizes the display model's content on the screen
Handler	modifies mathematical model after mouse and/or keyboard action
Updater	modifies slave MM-Object after dependant MM-Object has changed

The interaction between these components can be seen in the next figure which shows a simplified version, reduced to one single representative for each component's type.

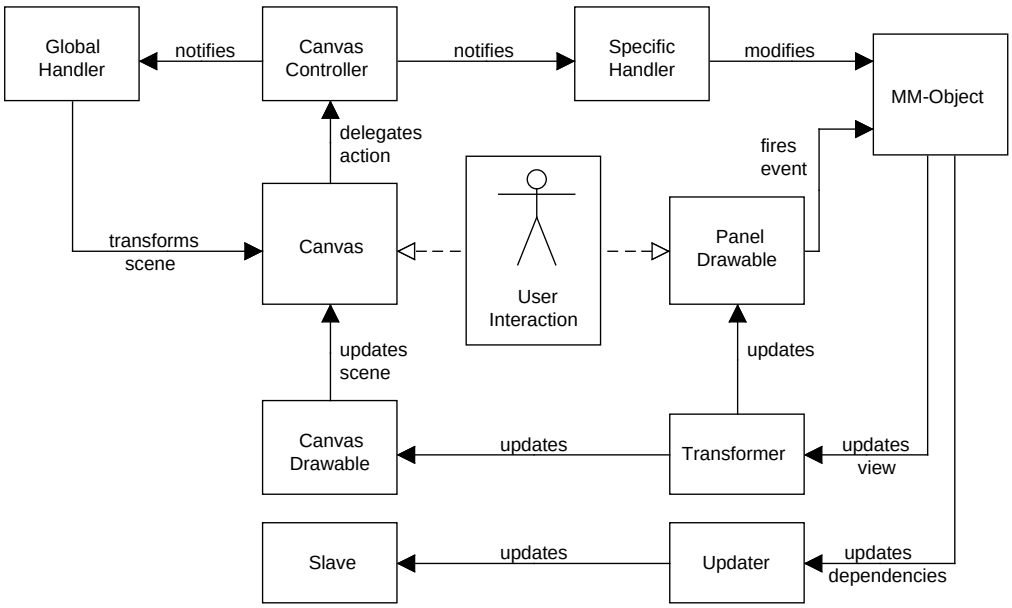


Figure 4: Interaction diagram of the MVC architecture

When the user modifies a graphical representation (e.g. drags a vector in a 2D canvas), the canvas propagates the AWT event (e.g. mouse drag event) to its canvas controller who generates a corresponding `MMEvent` and searches for an object specific handler: First it checks if a canvas object currently owns the focus and if such an object could handle the action. If one of these conditions is not met, the list of global handlers will be searched for a responsible handler instead. When a valid handler is found, it is supposed to react to the event by adjusting the mathematical model accordingly.

If any updater is associated with the underlying MM-Object its `doUpdate()` method will be called, in order to adjust the internal state of any dependant object which in turn might have other dependant objects and so on. Once all required operations are finished, every out-dated view will be redrawn (both graphical and symbolic representations).

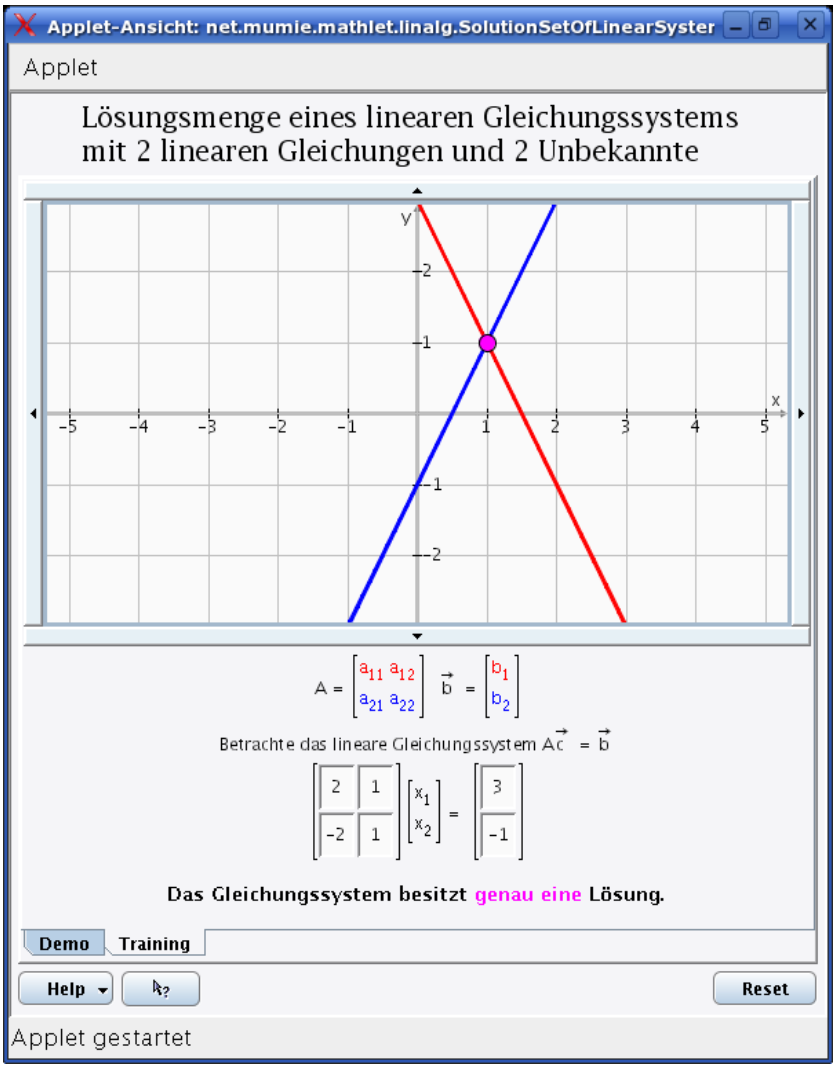


Figure 5: Screenshot: Solution Set of a Linear Equation System

4 Mathematical Models

4.1 Arithmetic Model

Since number fields and their corresponding properties play a fundamental role in numerical calculations, the MATHLETFACTORY uses *abstract arithmetics* for defining mathematical algorithms and properties and allows therewith to treat them independently of any number field and its precision. By using e.g. a rational number field, the precision of calculations can be freed of rounding errors, a common mathematical problem in computer sciences.
This technique is based on an abstract NUMBER CLASS (`MNumber`) which defines primitive arithmetic operations on an abstract number field.
The dependence of an abstract number field can be expressed by the interface `NumberTypeDependentIF`, which is also part of the general definition of MM-OBJECTS . Therefore every MM-OBJECT uses this abstract NUMBER CLASS instead of a concrete implementation.

M-Numbers

Every number field is represented by a single concrete NUMBER CLASS , starting with the prefix "M"[4]. As each of them is a mathematical model in the common sense of this document, they represent the fundament of every numerical calculation inside the MATHLETFACTORY .
The anonymous creation of number instances is done through the `NumberFactory`.

The most needed number classes integrated into the MATHLETFACTORY are:

MDouble	floating point numbers with double precision
MRational	rational numbers with integers
MOpNumber	operational numbers with internal number class
MInteger	whole numbers / integers
MComplex	complex numbers with doubles
MComplexRational	complex numbers with rationals

MM-Numbers

The number classes are fully included into the MVC architecture of the MATHLETFACTORY , i.e. both mathematical and multimedial objects are available.
Every number class listed above has a concrete MM-implementation (e.g. `MMDouble`) for which the same rules apply than for "usual" MM-OBJECTS , including display and interactivity support.

[4] This naming convention is due to the fact that the Java language also offers some "number" classes with the unprefixed name.

4.2 Algebraic Model

Mathematical entities are often defined by *algebraic expressions*, which may describe *operations* in a specific definition range or may even describe the definition range itself by defining *relations* between single expressions.
The MATHLETFACTORY uses an algebraic model which extends the arithmetic model and uses a FORMAL LANGUAGE for both *operations* and *relations*. These formal languages take advantage of the common syntax used in mathematics and thus allow the input of even complex content in a human-readable manner.

Evaluation of algebraic expressions

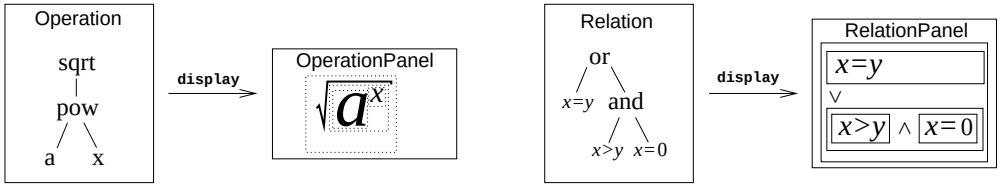
The analysis of a FORMAL LANGUAGE is processed at three levels of abstraction:

- the lexical analysis performs a regular pattern matching and replacements of symbols (e.g. "pi" $\rightarrow$ "π") and formal equivalences (e.g. "$2x$" $\rightarrow$ "$2 * x$")
- The syntactic analysis parses and maps the expression into an internal *operation / relation* tree with basically numbers and variables as leaf nodes
- the semantic analysis operates on the parsed tree and applies a set of *rules* transforming the tree into a normalized form

The resulting trees are either of the type `Operation` or `Relation` and offer additional features such as calculating the derivative and the definition range of an operation or applying custom transformation rules.
The numerical evaluation of a tree computes the result of each tree node up to the root node, thus building a recursive process.

Visualization of algebraic expressions

The tree structure of an `Operation` can be easily mapped on a new tree of *view nodes*, that draw the expression recursivly (see figure 6a [1]).
The single conditions of a `Relation` instead are mapped on a tree of *relation panels* whose leafs are *view nodes* again (cf. figure 6b [1]).



(a) Operations (b) Relations

Figure 6: View architecture of algebraic expressions [1]

4.3 Geometric Model

Though the MathletFactory supports different geometries, only affine geometric objects are implemented until now, using however homogenous coordinates internally. These entities allow calculations and transformations in well defined spaces and offer possibilities to visualize and change them both graphically and symbolically.

Geometric Groups

As the fundament for every of its geometries, the MathletFactory uses abstract *geometric groups* for which group operations must be defined (basically the composition). Implementations already exist for *affine* group elements and cover vector spaces, points, lines, polygons, angles, ...
Each affine group element implements the generic interface `AffineGeometryIF`, extending the abstract geometric group interface `GroupElementIF`.

Affine Spaces

As the fundament of all affine geometric objects, the entity `AffineSpace` represents an affine space with a finite *internal dimension* living in an environmental space with a finite *external dimension*. Thus a point in the $\mathbb{R}^2$ can be considered as an affine space with an internal dimension of zero and external dimension of 2, a line with an internal dimension of one and so on. In an analogous manner, every affine geometric entity in the MathletFactory is integrated as an affine geometry group element.

Projective Coordinates

In most Computer Aided Geometry (CAG) modelling software, all internal data is stored in homogenous (i.e. *projective*) coordinates, simplifying calculations in affine transformations. Therefore every `AffineSpace` uses projective coordinates internally and performs its actual computations in a *projective space*, represented by `ProjectiveSpace`. This entity uses internally a `NumberVectorSpace` with higher both internal and external dimensions and offers methods for working with both affine *and* projective coordinates.

4.4 Vector Space Model

Since calculations in linear spaces of higher dimension play an important role in mathematics, the MathletFactory defines a complete vector space model based on its arithmetic and geometric models and allowing to apply affine transformations from *domain* to *range* vector spaces by using the advantages of projective geometry.

Matrices and Tuples

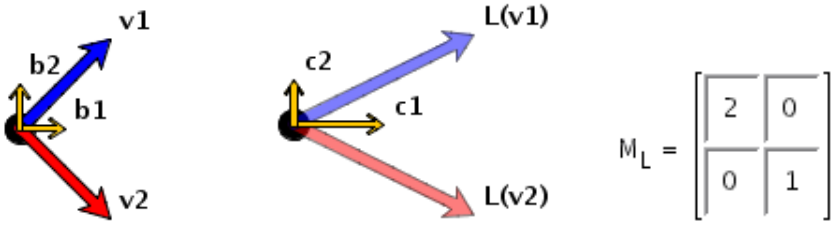
Multidimensional number computations are thereby performed by instances of `NumberMatrix`, a fundamental entity in this vector space model with an own MM-implementation for e.g. visualizing and editing arbitrary $m \times n$ number matrices. As a special extension for $m \times 1$ matrices, the entity `NumberTuple` is widely used for (vector) coordinates and matrix columns/rows, thus allowing to define additional features as calculating the norm or the dot product of vectors.

Basis Vectors

Each number vector (represented by `NumberVector`) is associated to a specific `NumberVectorSpace`, defining itself a basis and which the vector's coordinates are relative to. Thus changing the basis of a vector space allows to transform every vector contained in it. In order to visualize and edit both graphically and symbolically a vector space by its basis vectors, many MM-implementations (such as `MMDefaultRN` representing the $\mathbb{R}^n$) use this vector space entity.

Linear Transformations

Linear transformations or *linear maps* can either be expressed by their underlying map or by the definition of the vector spaces they connect. In the MATHLET-FACTORY the entity `LinearMap` can be used for this purpose, allowing to apply a linear map to a `NumberVector`. It uses internally instances of `NumberVectorSpace` for representing the *domain* and *range* vector spaces and stores its internal matrix representation into a `NumberMatrix`.



(a) Domain vector space (b) Range vector space (c) Symbolic view

Figure 7: Graphical and symbolic views of a linear map and two vectors

Again MM-implementations (such as `MMDefaultRNEndomorphism` representing an endomorphism in $\mathbb{R}^n$) exist and allow the visualization and editing of e.g. the map itself or the basis vectors of a vector space (cf. figure 7).

5 Homework Architecture

The MUMIE uses the powerful capabilities of the MATHLETFACTORY to load, edit and save complex homework data with mathlets and to correct and evaluate these with specialised *correctors*.
"Saving an exercise" means loading up the user's answers to the MUMIE server and getting a *receipt* for this saving transaction.

5.1 System Overview

The life cycle of every homework problem consists of basically three steps:

- the generation of a *data sheet* containing problem relevant, personalized data and already saved answers
- the execution of an exercise mathlet allowing to edit the answers
- the execution of an exercise corrector evaluating the answers

Whereas the mathlet runs on the client computer inside a Java Plug-In, the first and the last steps are executed on the server. The data exchange between the client and the JAPS server is done through a connection client (called JAPSCLIENT) using the MUMIE common XML data format (DATASHEET).

Exercise Mathlets

A mathlet loads the problem data and already saved answers from a MUMIE server (called JAPS – Java Application Server) and displays the values which also can be edited through this interface. Furthermore it allows to save newer answers to the MUMIE database.
See figure 8 for an overview of the exercise architecture in mathlets.

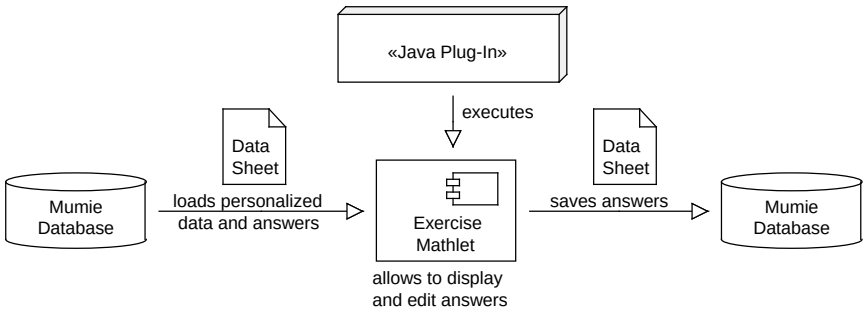


Figure 8: Exercise architecture in mathlets

Exercise Correctors

A corrector will be executed on the MUMIE server and allows an individual real-time "correction" with arbitrary complexness. They contain individual built-in algorithms and allow the donation for correct or particulary correct answers.
See figure 9 for an overview of the exercise architecture in correctors.

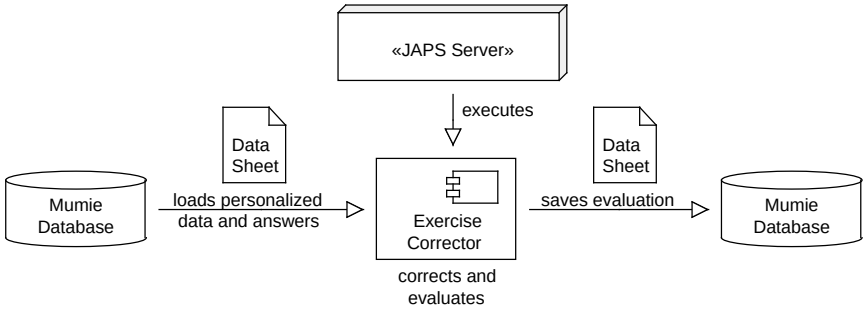


Figure 9: Exercise architecture in correctors

5.2 Homework Receipts

In order to have a "record" of the saved answers, the MUMIE server returns a *receipt* after the transmission, which includes a reference to the current problem and the answers encoded in MathML. These receipts can be used to view "offline" already saved answers on the server and to give a certainty about the correlation between the entered data and that on the server.
An included receipt viewer allows to generically view a receipt.

6 Outlook

In order to solve numerically e.g. even complex differential equations, a prospective objective is the connection to a Computer Aided System (CAS) for numerical evaluations which require a larger computing power or more sophisticated algorithms.

Another primary target is the development of powerful authoring tools which shall reduce the development and maintenance time for mathlets/correctors while increasing quality and compatibility. A first prototype based on the Java development environment OORANGE is available and allows an interactive and object oriented programming in a WYSIWYG manner of MUMIE mathlets and correctors, also known as *rapid prototyping*.

A further ambitious feature is the possibility to "script" exercise mathlets by defining the necessary algorithms for each problem without any knowledge about programming languages. This technique allows the simple creation of generic homework problems while using the capabilities of the MATHLETFACTORY in computing, visualizing and querying mathematical content. It is intended for problems whose complexity surpasses the possibilities of multiple choice exercises.

A last complex endeavour is the automatical generation of meta information and thus the semantical description of a mathematical program out of the composition of its components. This necessitates the design of an appropriate meta language, able to describe the ontology of mathematical problems, and the development of identification pattern for mapping mathematical entities and their relationships onto semantic structures.

References

[1] Dr. Tim Paehler, *Design, Implementation and Application of a Reusable Component Framework for Interactive Mathematical eLearning Sites*. Dissertation, RWTH Aachen, March 2005.

[2] Sun Microsystems, Inc., *Java Plugin Guide*. `http://java.sun.com/j2se/1.5.0/docs/guide/plugin/developer_guide/contents.html`.

[3] Sun Microsystems, Inc., *Java Deployment Guide*. `http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/contents.html`.

[4] Sun Microsystems, Inc., *The Swing Tutorial*. `http://java.sun.com/docs/books/tutorial/uiswing/index.html`.