# MAD : Model Aggregator eDitor

## Table of Content

## Motivations

Passionate about the industrialisation of software development, we have been designing tools for this domain for several years. We specifically work on communication protocol and application generators for languages such as RPG, Visual Basic, C, PHP and Java.

Originally our generators were based on specific internally-designed models. Interested by the arrival of new generation tools using EMF models, we chose to use UML for which an EMF metamodel already existed. This first step towards EMF then led us to adopt Ecore for all of our modeling requirements.

We have spent the last few months working on a new tool called ‘Verbose’. This is a framework designed to create generators which use EMF models. The main idea behind Verbose is to consider the templates used for the transformation of models into text as classes and to be able to combine them using design patterns adapted from the OOP. Our first use of Verbose was the creation of Verbose UML, a multilanguage application generator based on an UML conception which is independent of the target platform. We have therefore chosen to intensively use the different UML modelers available in the Eclipse environment. Most of them are fairly simple to use and allow us to draw diagrams quite easily. However, the interfaces for setting properties for our elements are not always easy to use. Some of these only provide the standard properties view, others offer a specific interface made up of widgets designed for inputting specific properties. These tools are therefore not particularly adapted to the user as the interface rarely provides the ideal information.

In order to simplify the editing process for our UML models, we have imagined the possibility of providing an ‘editing properties’ form that is configurable by the user. From the very first prototype designed for editing UML models we realised that the concept was applicable to any Ecore model. The MAD project, meaning Model Aggregator eDitor was then born.

## MAD Specification

MAD must propose a descriptive approach of the required editing form, by associating each property, which should be editable, with a type of widget. It must also be possible to describe the layout of these widgets.

The MAD editing form has to be reusable immediately after its description without code generation or compilation (runtime).

The labels on the MAD form have to be accessible to an international audience.

A validation mechanism with values entered by the user must be available before the EMF validation. Only validated values will be applied to the edited model.

The MAD form has to be compatible with any EMF model editor to edit the properties of the selected element.

It must be possible to edit elements from different models on the same form.

MAD will have to ensure a bidirectional synchronisation between the edited model and its editing forms. Any modifications carried out by MAD will have to be applied to the model; the MAD form will have to update itself when the model is modified by another editor.

MAD must allow the editing of any model element even without the use of an editor. It must be usable by any application or plugin.

## *Concepts*

### Configuring editing forms

The description of editing forms for a model is based on its Ecore metamodel. The structure and content of a form are described as a template associated with a type of metamodel element (EClass). The editing form for this kind of element will be built according to this description.

A template holds all of the required widgets; each widget must be specified by, at least, its type and a query expression which allows us to obtain the necessary value. The value of a widget will be obtained by the evaluation of its query from the element being edited. The reuse of pre-defined templates should be possible by inheritance and composition.

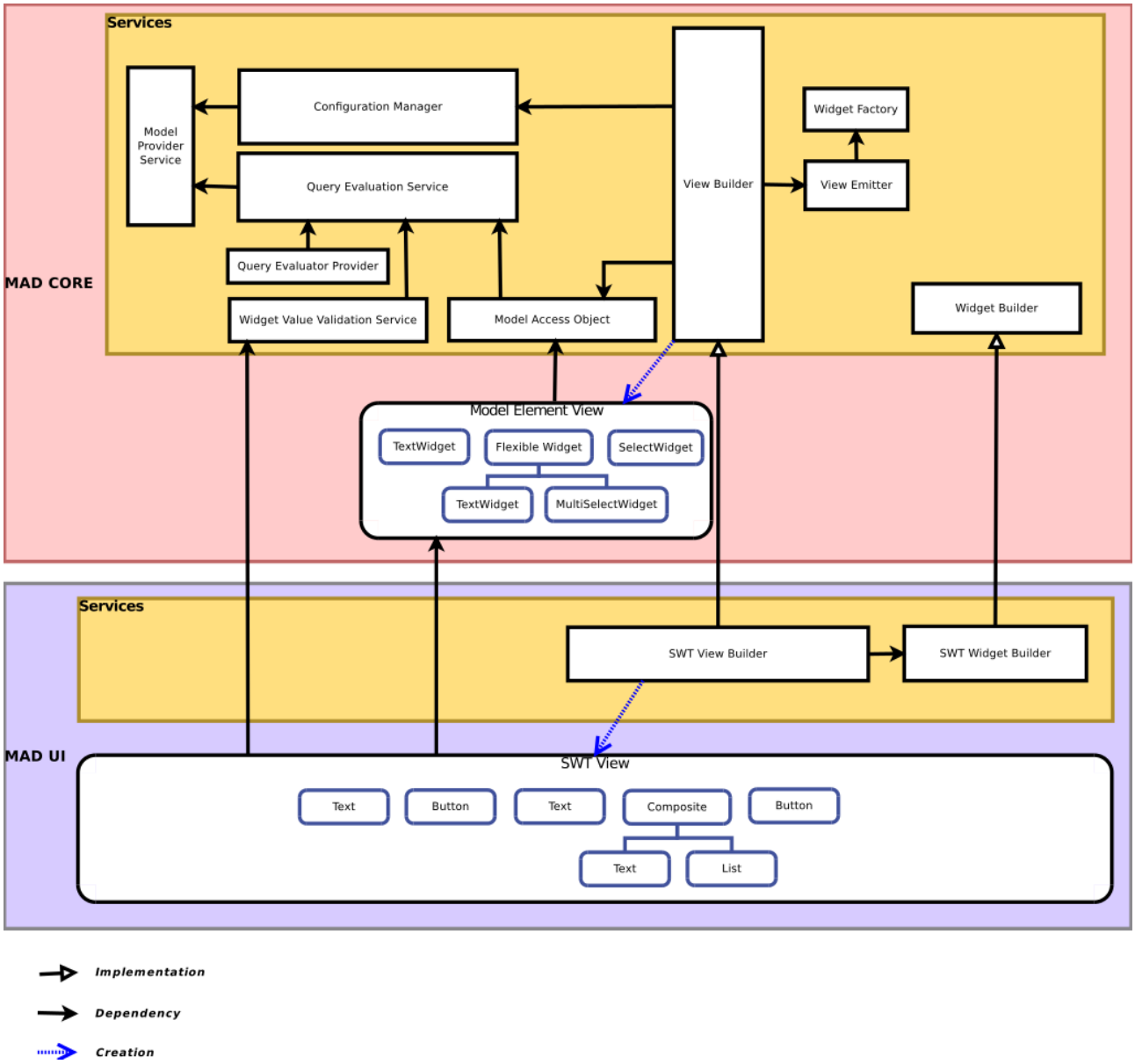
### Editing and backing up

Any modification of a value by means of a widget has to be applied immediately to the corresponding model property.

MAD handles two types of models, the main models: those that are already managed by a model editor and those known as ‘foreign’ models which are related to the main model; they are opened by MAD to allow them to be edited at the same time as a main model. The modifications of these model properties are carried out in a transactional way.

The **main model** properties modified by MAD are only updated in memory and the editor has to back up the model. The modification of **foreign model** properties implies MAD backing up these models.

## *Architecture*




**MAD is composed of three main Eclipse plug-ins :**

**API** : com.sysord.mad
**Core** : com.sysord.mad.core
**UI** : com.sysord.mad.ui

MAD is based on a Service-Oriented Architecture. The services' interfaces are defined in the *API* plug-in and all their default implementation are located in the *Core*. Adding or replacing services is possible thanks to the Guice module which can be extended through the extension point provided by the *Core* plug-in.

**Main services :**

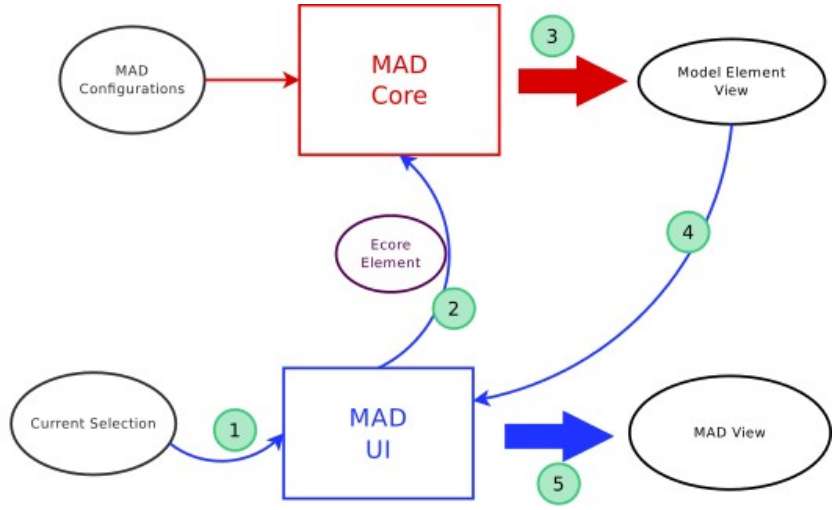
**ConfigurationManager** : Service for managing MAD configurations
**ModelProviderService** : Service providing EMF models.
**ModelAccessObject** : Service for reading and persisting elements in EMF models.
**QueryEvaluationService** : Service of query evaluation on EMF models.
**WidgetFactory** : Service for creating MAD widgets.
**ViewBuilder** : Service for creating the MAD view's content.
**WidgetBuilder** : Service for creating platform-specific widgets.
**WidgetValueValidationService** : Service of widget's value validation.

The *UI* plug-in uses MAD *Core* to create an **Eclipse view** with an edition form corresponding to the configuration and the EMF element currently selected in the Eclipse environment.

**Services implemented by the *UI* :**

**SWTViewBuilder** : Implements the **ViewBuilder** to generate the SWT view.
**SWTWidgetBuilder** : Implements the **WidgetBuilder** to generate the SWT specific widgets.

## Life cycle




***View Generation for an Ecore element***

## *Implementation choices*

MAD will obviously be an Eclipse plugin

## Configuration

The configuration of the MAD editing forms will be defined by a dedicated model. Given the mass of information that a configuration can involve and the expected readability, we have opted for a textual DSL created with Xtext.

From a description of the structure of a language, Xtext allows you to automatically generate the parser as well as an Ecore model representing the language elements and the relationships between them. It also produces an editor that is fully integrated into Eclipse and allows you to complete and validate syntactic structure.  A large number of extension points have been developed to add a series of features to the editor such as: content assist, specific validation rules, formatting strategies and many others. It is also possible to personalise the “text to model” step.

We have been using Xtext for many years now and we particularly like how quickly and easily we can create textual DSLs even if elaborating and integrating the DSL remain a project within a project

MAD allows you to edit any Ecore model, given that its configuration is an Ecore model; it would therefore be possible to use MAD as an editor of its own configuration

## *Integration into Eclipse*

The graphical aspect of MAD is an Eclipse view in which the widgets are constructed on the fly from the configuration associated with the element selected by a model editor. The widgets are therefore SWT widgets constructed using FormToolkit.

However, MAD is capable of creating widgets other than SWT; all you need to do is provide a personalised implementation of the services in charge of creating the graphical container for the view and add widgets to it (ViewBuilder and WidgetBuilder).

## Query evaluators

The widget values are obtained by evaluating queries from the selected element. We have opted for OCL as the main language but there is the possibility of creating and using personalised evaluators.

### The evaluators

As it is integrated into Eclipse, compatible with Ecore and extendable, OCL seemed to be the best choice for carrying out queries on the model.

OCL initially only provides the means to read in a model. When it is necessary to modify properties or create or delete elements, it is possible to invoke, from OCL, dedicated EOperations defined on the metamodel. UML provides a large number of these operations but not all models do so. In order to carry out these types of operations we have added a customised evaluator called MAD evaluator. It includes a set of basic functions allowing us to create, delete and move model elements. An **Acceleo** query evaluator for MAD is also available. It allows us to reference and use defined queries in the modules (.emtl) imported into the MAD configuration. We also tried integrating **EMF Query 2** but given the current state of progress of this project, the benefits of this language did not turn out to be useful so it was discarded.

### Spying on evaluators

The widgets are provided with the required values from reading queries. Any value assigned to an editable widget can be modified by the user and, in this case, the new value will have to be assigned to the corresponding property in the model. It is then necessary to find out which property has provided the resulting value of the query. In order to do this we have used two methods for ‘spying on evaluators’ and determining the origin of the results they provide.

- The first, specialised for OCL, consists in intercepting the property access methods and gathering contextual information. It is therefore possible to obtain the property that has been accessed and its owner. Towards the end of the evaluation a validation of this information is carried out by comparing the result of the interception and that of the evaluation. If these do not match, the evaluation analysis is considered invalid. This solution has a good success rate for simple queries but is inefficient when handling queries using model element operations or advanced OCL functions.

- The second solution is meant to be generally applicable to any evaluator; it consists in instrumenting the basic element of the query, the context, by the dynamic proxy mechanism. This proxy intercepts the operations invoked on the element in order to collect the accesses to its properties; when the type of result permits, a proxy with the same behaviour is returned. At the end of the evaluation, a validation of this analysis is carried out. This method compensates for some of the features that are missing in the previous solution but still present various limitations:

- Some of the ‘uninstrumentable’ types play an important role in the evaluation but are not monitored

- The internal evaluator behaviour can result in unmonitored structures or inaccurate results when comparing the proxies.

We are currently working on improving these solutions.

When the query evaluation analysis fails, MAD cannot determine which action to carry out to update the value. The action can then be explicitly provided in the configuration by associating the adapted query in UPDATE_COMMAND with the widget.

## Synchronising models and views

This consists in ensuring that any changes made by MAD to a model element are immediately taken into account by the model editing view. In the same way, any changes made by the model editor must be applied to the MAD view.

The synchronisation of the model editor view is managed by the model editor but is not the responsibility of MAD. In order to guarantee that the MAD view reflects the model image at any given time we have used the adapters provided by EMF. Registered adapters on the edited element notify MAD of any changes; the corresponding view widgets are updated.

## Dependency injections

MAD's architecture is based on a set of services, each one designed to manage part of the necessary processing flow and the view for an element that you will edit. In order to link these services used by MAD, we have resorted to use dependency injections which ensure a loose link between components. Each service is presented as an interface; there can be one or several different implementations for each service. A configuration defines which implementation will be used for all services upon execution. A MAD user can provide their own implementation for each service.

The initial version of MAD was developed for Eclipse Indigo; for this reason, we were not able to use the Eclipse 4 dependency injection mechanism and therefore opted for Guice.

## *Queries*

## OCL Evaluator

The OCL evaluator uses standard syntax. We added various applicable operations to all of the elements in order to simplify query writing.

**`toString(context : OclAny) : String`**

*Returns the String equivalence of the* `context`*. Behind the scenes, the Java toString() method is invoked on the* `context`*.*

**`rootContainer(context : OclAny) : EObject`**

*Returns the root element of the given* `context`*'s model if it's an EObject,* `null` *otherwise.*

**`eContainer(context : OclAny) : EObject`**

*Returns the container of the given* `context` *if it's an EObject,* `null` *otherwise.*

## MAD Evaluator

The MAD evaluator offers various functions for manipulating model elements.

**`CREATE(context:EObject, FeatureName:String, eClassName:String, container:EObject)`**
*Creates a new element and puts it in its corresponding container. Returns the created element.*

**Parameters :**

**context** : *The* `context`*. If the given container is* `null`*, the* `context` *element is considered as the container.*

**FeatureName** : *The name of the feature that will contain the created element.*

**eClassName** : *The EClass' name of the element to create.*

**container** : *(optional) Container of the element to create.*

```
UPDATE_FEATURE(context:EObject, featureName:String, value:Object)
```

*Modifies the value of an element's property.*

**Parameters :**

**context** : *The context of the element for which the property is modified.*

**featureName** : *The name of the feature to modify.*

**value:** *The value to set.*

```
DELETE(context:EObject)
```

*Deletes an element from its model. The element, all its children and all their references are deleted.*

*The deletion is done with* **EcoreUtil.delete(context, true)**.

```
REMOVE(context:EObject)
```

*Removes an element from its container.*

*The removal is done with* **EcoreUtil.remove(context)**.

```
MOVE_UP(context:EObject), MOVE_DOWN(context:EObject)
```

*Move, up or down, an element contained in a collection-based feature.*

## Query and sub-query chains

MAD queries can be made up of several queries, each written in a different language. This kind of query writing allows us to combine the functions of the different evaluators in order to make the most of each one and simplify complex query making.

### *Elementary query*

A simple MAD query contains the body of the query in the form of a string. The language identifier must be provided if the query is not written in OCL.

```
"authors"
```

*Returns the authors of a book. No defined language, OCL is used by default.*

```
Language: ACCELEO call authorsOfSeveralBooks()
```

*Call of an Acceleo query. This query returns the authors of the book who have written other books.*

```
Language: MAD "CREATE('books')"
```

*Call of a MAD function which creates a book element into the library.*

### *Query chain*

The MAD query chain is made up of a list of organised elementary queries. The queries are evaluated one after another, the result of a query determining the context for evaluating the next.

```
Query Chain {
     "books->last()",
     language: ACCELEO call authorMultiBook(),
     "first()"
}
```

*Query chain with queries of different languages (OCL and Acceleo). This query chain returns the first author of the last book in the library who wrote several books present in the library.*

### Sub-queries

A main query can include sub-queries which are written between square brackets. The evaluator starts by evaluating the sub-queries from the context; the results obtained are injected into the main queries in order to replace the sub-query.

```
eContainer().oclAsType(Library).books->select(pages > [pages])->isEmpty()
```

*Returns true if the book is the one with the most pages in the library. The first '*`pages`*' refers to the books in the iteration. '*`[pages]`*' is evaluated before the iteration and corresponds to the number of pages of the book.*

### Contextual variables

In order to simplify writing queries and make them more explicit, a set of predefined variables are available. These variables, set by MAD before the evaluation, allow us to access the view context, the value of the edited element and other contextual information.

```
$UIVALUE > 10 and $UIVALUE < 10000
```

*Validation rule of a numerical widget's value.* `$UIVALUE` *is substituted by the value written in the widget.*

### Customised evaluators

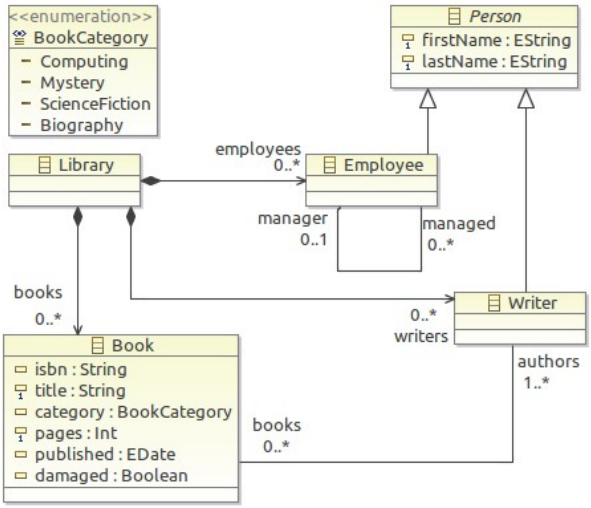
In order to meet specific needs, MAD allows you to create your own evaluators, reference them in the configuration and use them for evaluating queries. A prefixed expressions parser is available and can easily be reused by any new language.

A customised evaluator is created by realizing the interface **QueryEvaluator**.

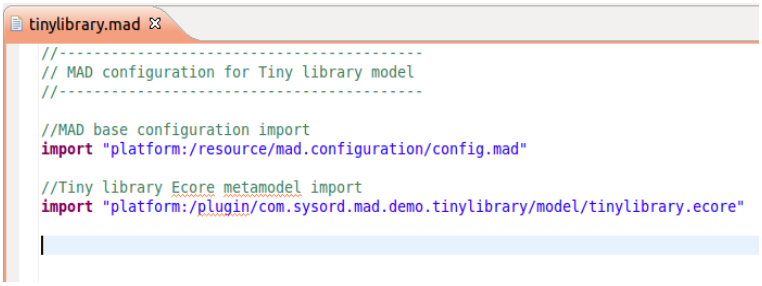
## MAD in action

The first step for using MAD consists in creating a configuration in which we define the required editing views for the model we want to edit. All we need to do is add this configuration to the MAD preferences and that’s it, the property editor is ready. All the configuration changes will be taken into account immediately.

*The following examples will use the 'tinylibrary' model which is a simplified adaptation of the Library model.*




Creation of a MAD configuration for the 'tinylibrary' models. At this point no editing view has been defined.




## *Widgets*

MAD provides a range of configurable widgets for creating customised forms.

**Editing widgets:**

- Entering Text, Number, Date and Boolean values
- Selecting values: Combobox, integrated or popup Lists and PickLists
- Editing Xtext model elements (using the integrated Xtext editor)

**Widgets for viewing purposes**: Read-only display of a calculated or elementary value.

**Widgets for browsing purposes**: list of proposed elements allowing direct access to their detailed view.

**Flexible widget**: Displaying a list of editable elements, each represented by the set of widgets described in its corresponding template.

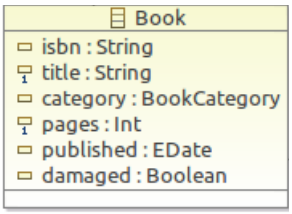
**Composite widget**: Including all the widgets of a predefined template in a given template in order to represent a composite element for which we want to homogenise the rendering of each of its occurrences

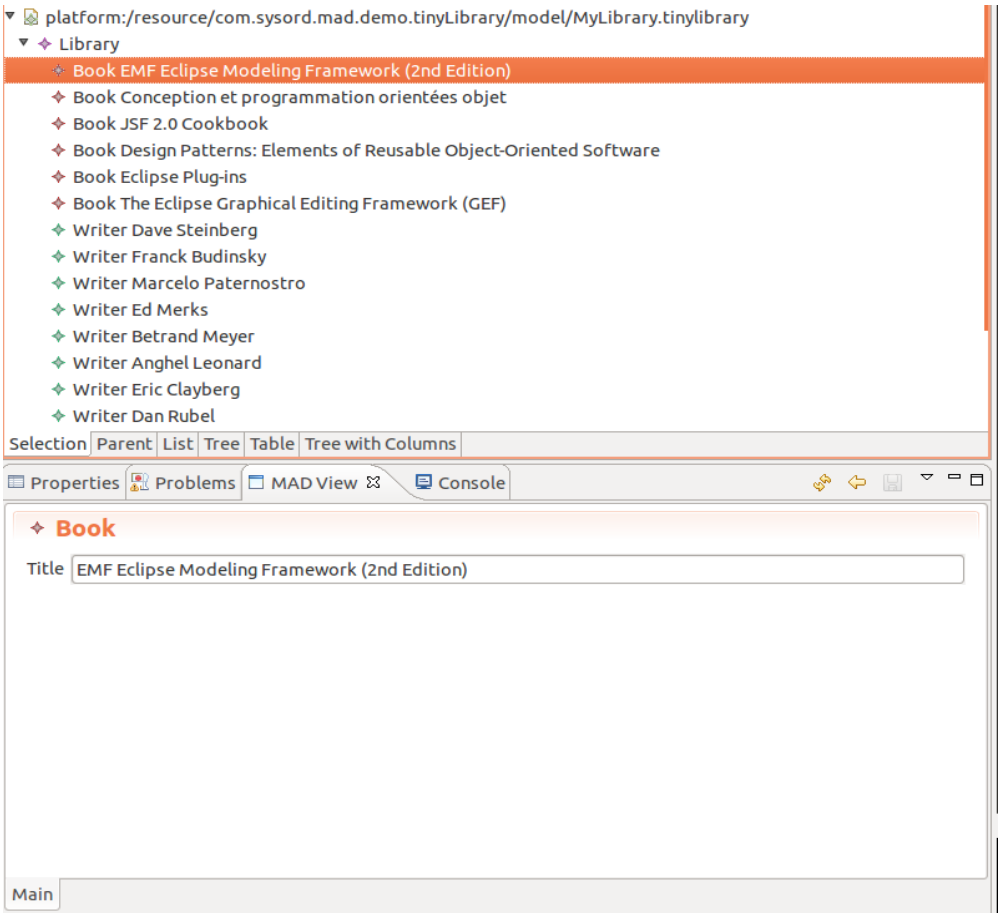
**Command**: The button for launching actions.

The range of widgets provided by MAD is fully extendable; it's possible to modify each original widget by creating its own implementations.

## A First Configuration

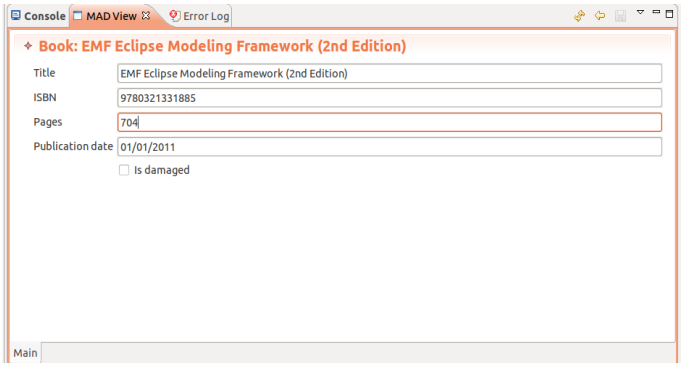
```
//-----------------------------------------
// MAD configuration for Tiny library model
//-----------------------------------------

//MAD base configuration import
import "platform:/resource/mad.configuration/config.mad"

//Tiny library Ecore metamodel import
import "platform:/plugin/com.sysord.mad.demo.tinylibrary/model/tinylibrary.ecore"
//Configuration for a Book element
Configuration BOOK for tinylibrary.Book {
    template:

    //Textbox widget for editing title property
    widget:Title    //the widget id
    label:"Title"   //widget label
    type:TEXT_WIDGET          //display a text widget
    value:"title"   //Ocl query for getting the 'title' property from the book.
}
```



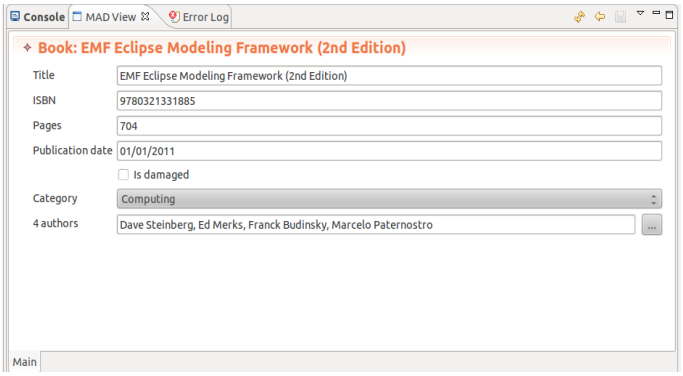
## Simple Widgets

```
Configuration BOOK for tinylibrary.Book {
    //Format expression for all Book elements Label computing
    //queries between [] are evaluated parts.
    label provider:"Book: [title]"

    template:
     ...
    //Textbox for isbn
    widget:Isbn
    label:"ISBN"
    type:TEXT_WIDGET
    value:"isbn"

    //Number input widget
    widget:Pages
    label:"Pages"
    type:NUMBER_WIDGET
    value:"pages"

    //Date input widget
    widget:PublicationDate
    label:"Publication date"
    type:DATE_WIDGET
    value:"published"

    //Checkbox widget
    widget:Damaged
    label:"Is damaged"
    type:BOOL_WIDGET
    value:"damaged"

}
```



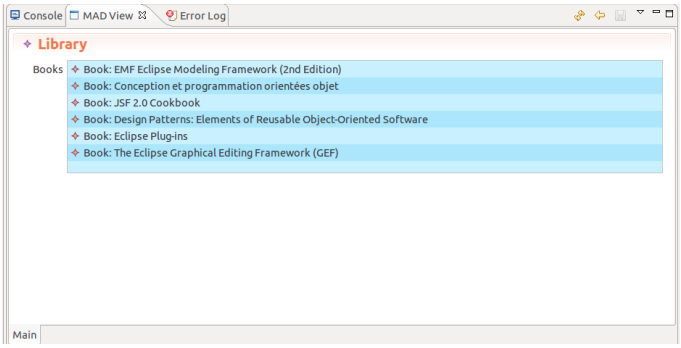

## List Widgets

```
 ...

    //Combo widget
    widget:Category
    label:"Category"
    type:SINGLE_SELECT_WIDGET
    value:"category":tinylibrary.BookCategory
    //OCL query for filling combo
    candidates:"BookCategory.allInstances()"

    //Popup PickList widget
    widget:Authors
    //Dynamic label value
    label:"[authors->size()] authors"
    type:MULTI_SELECT_WIDGET:POPUP_PICKLIST
    value:"authors"
    //Populate the list with candidates query results
    candidates:"eContainer().oclAsType(Library).writers"
    item label:"[name]"
```




## Navigation Widget

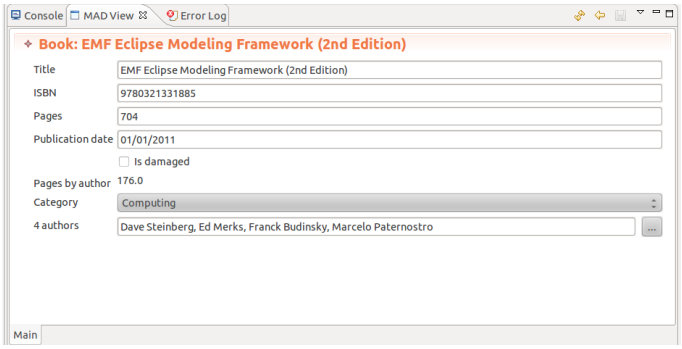
```
//Configuration for the Library element
Configuration LIBRARY for tinylibrary.Library {

    template:
    //Navigation widget for accessing Book detail
    widget:BooksNavigation
    label:"Books"
    type:NAVIGATION_WIDGET
    candidates:"books"
}
```




The navigation widget proposes a list of elements. A double-click on one of them allows to navigate toward the detailed view of this element. The left arrow allows to go back to the previous view.

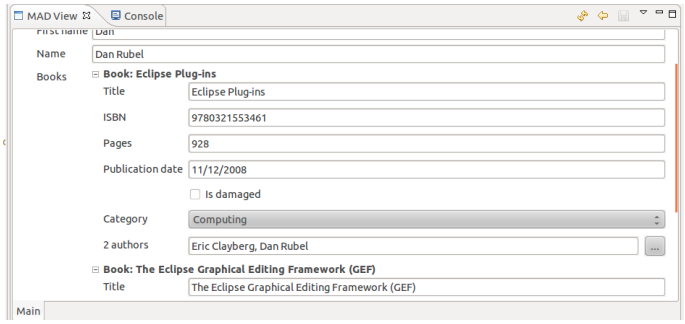
# Widget for displaying Read-Only values

```
Configuration BOOK for tinylibrary.Book {

     ...
    //Output text
    widget:avgPage
    label:"Pages by author"
    type:OUTPUTTEXT_WIDGET
    //conditional visibility
    visible when:"not authors->isEmpty()"
    //Compute pages average by author.
    value:"(pages / authors->size())"
    //value converter from double to string.
    valueConverter:DOUBLE
}
```



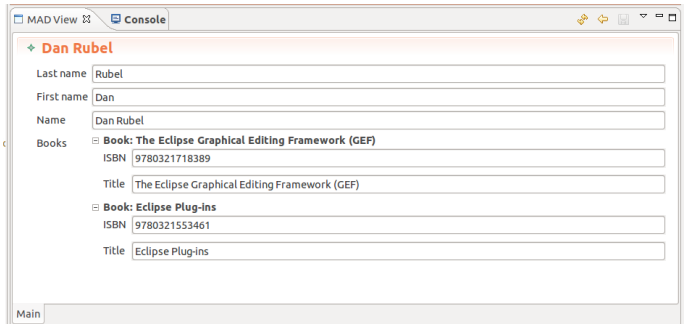

# Flexible Widget

```
//Configuration for a person element (abstract)
Configuration Abstract_PERSON for tinylibrary.Person {

    label provider:"[name]"
    template:

    widget:Name
    label:"Name"
    type:TEXT_WIDGET
    value:"name"

    widget:FirstName
    label:"First name"
    type:TEXT_WIDGET
    value:"firstName"

    widget:LastName
    label:"Last name"
    type:TEXT_WIDGET
    value:"lastName"
}

//Configuration for a Writer element
//extends implicitly Person configuration
Configuration WRITER for tinylibrary.Writer {
    template:

    widget:Books
    label:"Books"
    type:FLEXIBLE_WIDGET
    //include Book template for each writen book
    value:"books"
}
```



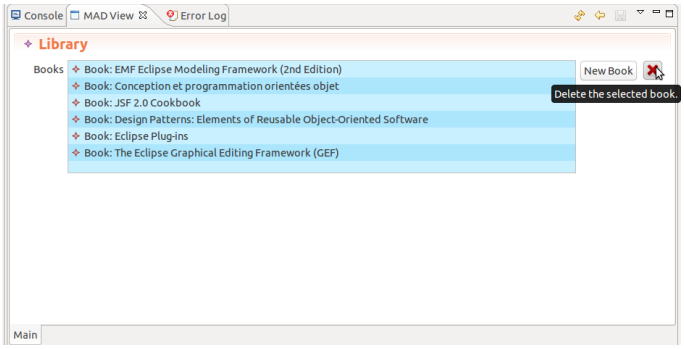

# Selection of a specified template for the flexible widget elements

```
//Alternative configuration for a Book element
Configuration BOOK_SHORT for tinylibrary.Book {

    //Explicit extension
    extends: BOOK //Reuse the BOOK template
    template:
    //Display only those widgets
    layout: Isbn Title
}

Configuration WRITER for tinylibrary.Writer {
    template:

    widget:Books
    label:"Books"
    type:FLEXIBLE_WIDGET
    //Use the BOOK_SHORT template
    flexible template: BOOK_SHORT
    value:"books"
}
```



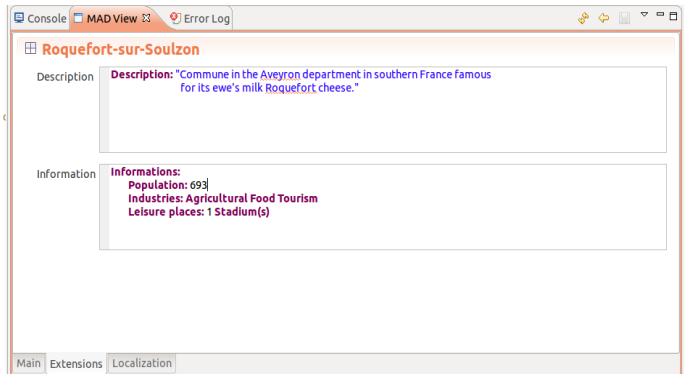
## Widget Command

```
//Icon declaration
Use icon DELETE_ICON URI:"platform:/resource/mad.configuration/icons/delete-icon_16.png"

//Shared command declaration
Common Command DELETE_ELEMENT_COMMAND {
    ITEM_COMMAND
    "Delete item"      //Command label
    icon:DELETE_ICON   //Image for the command button
    //launch the DELETE MAD Macro for deleting selected item
    action: language:MAD "DELETE()"
    on success: Reload view
}

Configuration LIBRARY for tinylibrary.Library {

    template:
    widget:BooksNavigation
    label:"Books"
    type:NAVIGATION_WIDGET
    candidates:"books"
    commands:
      //Inner command for creating a new book
      GLOBAL_COMMAND "New Book"
      action: language:MAD "CREATE([OCL:'books'])"
      //after creation displays view
      //for the created item: the command RESULT.
      on success: Display view for "$RESULT",
      //Use shared command with label override
      DELETE_ELEMENT_COMMAND("Delete the selected book.")
}
```




## Other widgets

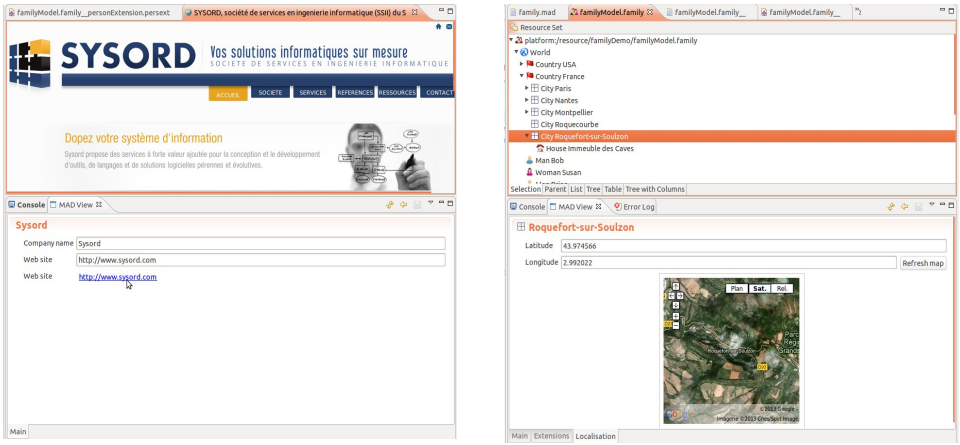
### *Xtext editor*




The Xtext editor widget allows you to add integrated editors to the MAD view in order to modify Xtext model elements using completion and validation.

Developing this widget caused us a few problems due to the way that Xtext functions. Any modification in the text will launch the parser and reconstruct the entire branch of the impacted model. For this reason, it is not worth keeping references or placing Adapters on Xtext model elements for which the sub-elements are likely to change as these references can become obsolete. Modifying either of the texts in the above example deletes the other's model. Keeping a copy of the edited element in memory as well as its URI should a modification merge occur seems to be a good compromise. However, if the element's URI is not based on a unique identifier or if there has to be links (references, validation rules) between the two elements, editing or merging will not be carried out correctly. It is therefore necessary to reload all the potentially impacted Xtext widgets found on the view when one of them modifies the model.

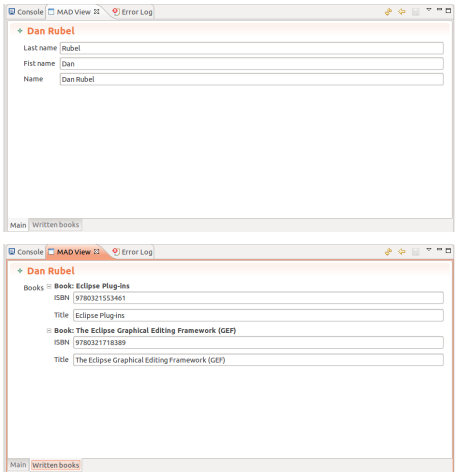
### *Html link and Google MAP widgets*




These widgets have been designed by customising the **OUTPUTEXT_WIDGET**. Their detailed configuration is presented in a video of the first version of MAD: MAD is Customizable

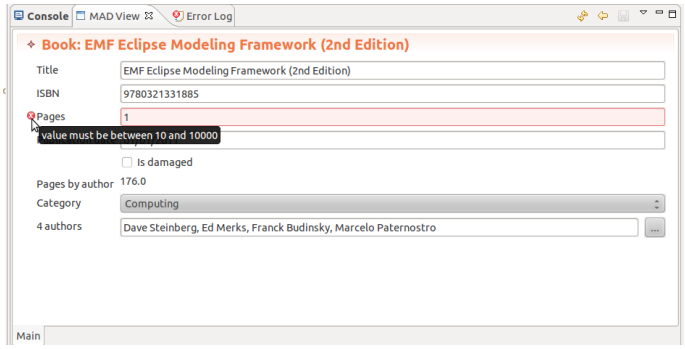
## Layout

```
//tab declaration
UI Tab {
    id:WRITEN_BOOKS
    label:"Writen books"
}

Configuration WRITER for tinylibrary.Writer {
    template:

    widget:Books
    //the widget will be diplayed on the WRITEN_BOOKS tab
    tab:WRITEN_BOOKS
    label:"Books"
    type:FLEXIBLE_WIDGET

    flexible template: BOOK_SHORT
    value:"books"

    //widgets display order definition
    layout: LastName FirstName Name Books
}
```




## Validators

```
Configuration BOOK for tinylibrary.Book {

    ...

    //Number input widget
    widget:Pages
    label:"Pages"
    type:NUMBER_WIDGET
    value:"pages"
    validators:
    //Validation: pages widget must be filled
    //and its value between 10 and 10000
    validation rule:"not $UIVALUE.oclIsUndefined()"
      I18N Error message:"REQUIRED_VALUE"
    validation rule:"$UIVALUE > 10 and $UIVALUE <
10000"
      I18N Error message:"VALUE_OUT_OF_RANGE[10]
[10000]"
    ...
}
```

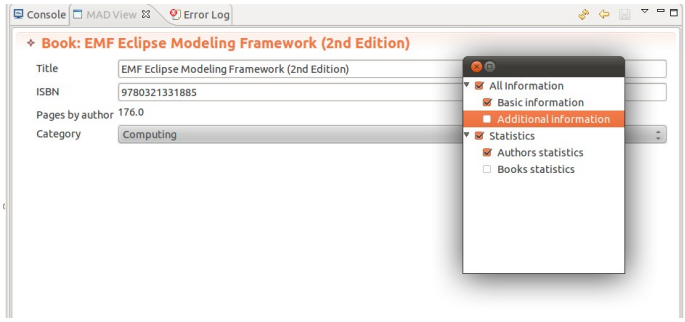



Validation rules placed on the widgets can verify the conformity of the values before updating the model.

## Layers

The layers allow you to collect and select the widgets that you want to display depending on a particular theme. These are useful as they allow you to gather the widgets with information from the same domain. These layers are stackable and hierarchical. They allow you to refine the view according to the user's requirements.

```
//Layers configuration
Layer INFO_LAYER {
    label:"All Information"
    Sub layers {
          Layer BASIC {label:"Basic information"}
          Layer DETAILED{label:"Additional
information"}
    }
}
Layer STATISTICS_LAYER {
    label:"Statistics"
    Sub layers {
          Layer AUTHOR{label:"Authors statistics"}
          Layer BOOKS{label:"Books statistics"}
    }
}
Configuration BOOK_SHORT for tinylibrary.Book {
    ...
    widget:Pages
    //available in books statistics layers
    layers: STATISTICS_LAYER.BOOKS
    ...
    //Output text
    widget:avgPage
    //available in the two layers
    layers:STATISTICS_LAYER.BOOKS,
          STATISTICS_LAYER.AUTHOR
    label:"Pages by author"
    ...
}
```




## I18N

MAD supports internationalisation by using Eclipse NLS features (National Language Support). The translations are stored in properties files for each locale; a symbolic name is assigned to a translation in the target locale. In the MAD configuration, calculated or elementary texts that we want to translate are preceded by the I18N keyword and their value contains the symbolic name for the required translation followed by queries (optional) designed to calculate and provide the variable parts (parameters) of the translation. MAD provides translated labels for the system's default locale. All the MAD configuration labels are internationalisable.

## Utility of MAD and use cases

MAD allows personalised editing for any Ecore model; it is therefore a tool that can be used in all MDE and MDA approaches applied in the Eclipse environment. The creation and maintenance of models is a very important task that has to be carried out with maximum user comfort. MAD improves the user's experience by providing them with editors adapted to their needs or a complement to the modelers and editors that they use.

Thanks to its dynamic aspect, MAD can allow the rapid creation of a user interface by following the user's instructions. This kind of design shares many similarities with the RAD (Rapid Application Development) approach in which a user describes their needs; the designer configures a generator and the result is immediately evaluated by the user. Through rapid iterations, the user obtains the expected product. With MAD there is no generation phase; any configuration modification is immediately visible.

The user describes their ideal interface. The designer only needs to have a basic knowledge of how to write queries and a good knowledge of the target model. The designer configures MAD according to the description of the user's needs; the user immediately sees the result, gives feedback and either agrees on the result or asks further modifications.

Another useful MAD feature allows you to create an identical editing mode for a single element, whatever the modeler or model editor. Thus, editing a UML element with the Eclipse UML model editor, Papyrus or UML designer will be easier as all three are highly similar, which will also reduce the adaptation time for the user to get started on a new modeler.

MAD allows you to simultaneously edit elements that come from several different models; it is fully adapted for model extension or decoration. A main model holds the main information for the edited domain; a complementary model holds additional information which is needed for specific uses. MAD allows you to aggregate this information on a given editing view and modify or simultaneously consult the two models. Using layers allows you to refine the view following this merge.

When carrying out a decoration, for each element of the main model there is usually a corresponding counterpart in the decorator model. One of the underlying issues with this type of editing is the need to synchronise the two models: creating any element that you want to decorate in the main model implies creating its counterpart in the decorator model. MAD offers dedicated services for simplifying model extension and decoration.

**ModelExtensionManager** is a service used by the MAD_EXTENSION query language. MAD_EXTENSION allows you to write queries to retrieve, from the decorator or extension model, the element which corresponds with a main model element. All you need to do is provide an implementation of the **ModelExtensionManager** interface which implements the correspondence strategy that you need to adopt to establish a link between a main model and an extension or decorator model.

**ExtensionModelSynchroniser** is a service used for synchronising the main model with an extension model. This is a main model observer which monitors all element creation and deletion and uses the **ModelExtensionManager** to transmit and apply modifications to the extension model. The implementations to use for a model are defined in the MAD configuration.

(video demos: Family Model Extension, MAD with Xtext)

Finally, MAD is extendable and open; it allows you to easily adapt to new needs as soon as they are identified, either by configuration or extension: integration or development of new specific query languages, creation of personalised view generators and widgets for a new graphical environment, reuse of the MAD core through applications.

## *Conclusion*

MAD is now operational; we use it internally at Sysord for designing and editing all of our models and more specifically for UML models used as input for application generators (demo video: MAD with UML). A number of evolutions, including a query compiler, are currently being developed to improve the performance of MAD and make it easier for users to configure. We are also planning on developing many others in the future:

- Integration of CDO in order to manage the multiuser mode and concurrent access
- MAD configuration generator: from a selected element for which no template has been defined, a button allows you to produce a default description through element introspection and automatically add it to the current configuration.
- Possibility of creating commands which invoke Java methods through reflection.
- MAD editor generator: from a tried and tested configuration in interpreted use, generation of an Eclipse plugin
- Editing multimedia elements (sound, image, video, charts, graphs, ...)
- Configuration environment that provides wizards for creating queries
- Using Eclipse 4 and why not JavaFX features