MAD : Model Aggregator eDitor

Table des matières

MAD : Model Aggregator eDitor....1
- Motivations :....2
- Spécification de MAD :....3
- Concepts : 3
  - La configuration des formulaires d'édition :....3
  - Edition et sauvegarde:....3
- Architecture :....4
  - Cycle de vie :....5
- Choix pour l'implémentation :....5
  - Configuration :....5
- Intégration à Eclipse :....6
  - Évaluateurs de requêtes :....6
  - Synchronisation Vues modèle :....7
  - Injection de dépendances :....7
- Requêtes:....8
  - Evaluateur OCL :....8
  - Evaluateur MAD :....8
  - Evaluateur MAD EXTENSION:....9
  - Chaînes de requêtes et sous-requêtes :....9
    - Requête élémentaire :....9
    - Chaîne de requêtes :....9
    - Sous-requêtes :....10
    - Variables contextuelles :....10
    - Evaluateurs personnalisés :....10
- MAD en action :....11
- Widgets :....11
  - Une première configuration :....12
  - Widgets simples :....13
  - Widgets Listes :....13
  - Widget pour la navigation :....13
  - Le widget de navigation propose une liste d'éléments. Un double-clic sur un élément permet de naviguer vers sa vue détaillée. La flèche vers la gauche permet de revenir à la vue précédente....13
  - Widget pour affichage en lecture seule:....14
  - Flexible widget :....14
    - Sélection d'un template particulier pour les éléments du flexible :....14
  - Widget commande :....15
  - Autres widgets :....15
    - Editeur Xtext :....15
    - Html Link et Google MAP widgets :....16
  - Layout :....16
  - Validateurs :....16
  - Des règles de validations placées sur les widgets permettent de vérifier la conformité des valeurs avant la mise à jour du modèle. 16
- Layers : 17
- I18N :....17
- Intérêts de MAD et cas d'utilisation:....18
- Conclusion :....20

Motivations :

Passionnés par l'industrialisation du développement logiciel, nous concevons depuis un certain nombre d'années des outils pour ce domaine. Nous réalisons plus particulièrement des générateurs d'applications et de protocoles de communication pour des langages aussi variés que RPG, Visual Basic, C, Php, Java.

A l'origine, nos générateurs se basaient sur des modèles « maison » spécifiques. Séduits par l'arrivée de nouveaux outils de génération exploitant les modèles EMF nous avons choisi d'utiliser UML pour lequel il existait un métamodèle EMF. Ce premier pas vers EMF nous a ensuite conduits à adopter Ecore pour tous nos besoins en modélisation.

Nous travaillons depuis quelques mois sur un nouvel outil nommé Verbose. Il s'agit d'un framework dédié à la création de générateurs exploitant des modèles EMF. L'idée principale de Verbose est de considérer les templates utilisés pour la transformation de modèle vers du texte comme des classes, et de pouvoir les combiner entre eux en utilisant des design patterns adaptés de la POO. Notre première utilisation de Verbose a été la création de VerboseUML, un générateur d'applications multi-langages basé sur une conception indépendante de la plate-forme cible réalisée en UML. Nous avons donc utilisé de manière intensive les différents modeleurs UML disponibles dans l'environnement Eclipse.

Ils sont pour la plupart simples d'utilisation et permettent de construire assez facilement des diagrammes. Par contre, les interfaces pour l'alimentation des propriétés de nos éléments ne sont pas toujours pratiques d'utilisation, certains ne mettent à disposition que la vue standard des propriétés, d'autres fournissent une interface spécifique composée de widgets destinés à saisir des propriétés particulières. Ces outils sont donc peu adaptés aux besoins de l'utilisateur car l'interface comporte rarement les informations idéales pour celui-ci.

Pour faciliter l'édition de nos modèles UML, nous avons imaginé la possibilité de fournir un formulaire d'édition des propriétés qui soit totalement configurable par l'utilisateur. Dès le premier prototype dédié à l'édition des modèles UML, nous nous sommes rendu compte que le concept était applicable à tout modèle Ecore : le projet **MAD** pour **Model Agregation eDitor** commençait.

Spécification de MAD :

- MAD doit proposer une **approche descriptive** du formulaire d'édition à obtenir, en associant chaque propriété à rendre éditable à un type de **widget**. La disposition de ces widgets doit également pouvoir être décrite.
- Le formulaire d'édition MAD doit être utilisable immédiatement après sa description **sans génération de code ni compilation** (Runtime).
- Les **labels** présents sur un formulaire MAD doivent être **internationalisables**.
- Un mécanisme de **validation des valeurs entrées** par l'utilisateur doit être disponible en amont de la validation EMF. Seules les valeurs validées seront appliquées au modèle édité.
- Le formulaire MAD doit pouvoir être utilisé avec **n'importe quel éditeur de modèle EMF** pour éditer les propriétés de l'élément sélectionné.
- Il doit être possible d'éditer sur un même formulaire des **éléments provenant de différents modèles**.
- MAD devra assurer une **synchronisation bidirectionnelle entre les modèles édités et ses formulaires d'édition**. Toute modification effectuée par MAD devra être appliquée sur le modèle, le formulaire MAD devra s'actualiser lorsque le modèle est modifié par un autre éditeur.
- MAD doit permettre l'édition de tout élément de modèle **même sans l'utilisation d'un éditeur**. Il doit pouvoir être utilisé par n'importe quelle application ou plugin.

Concepts :

La configuration des formulaires d'édition :

La description des formulaires d'édition pour un modèle se base sur son **métamodèle Ecore**. La structure et le contenu d'un formulaire est décrit sous la forme d'un template associé à un type d'élément (**EClass**) du métamodèle. Le formulaire d'édition d'un élément de ce type sera construit d'après cette description.

Un template comporte l'ensemble des widgets à fournir, chaque widget est spécifié au minimum par son type et une expression requête permettant d'obtenir la valeur qu'il doit présenter. La valeur d'un widget sera obtenue par l'évaluation de sa requête à partir de l'élément en cours d'édition.

La réutilisation de templates déjà définis devra être possible par héritage et composition.

Edition et sauvegarde:

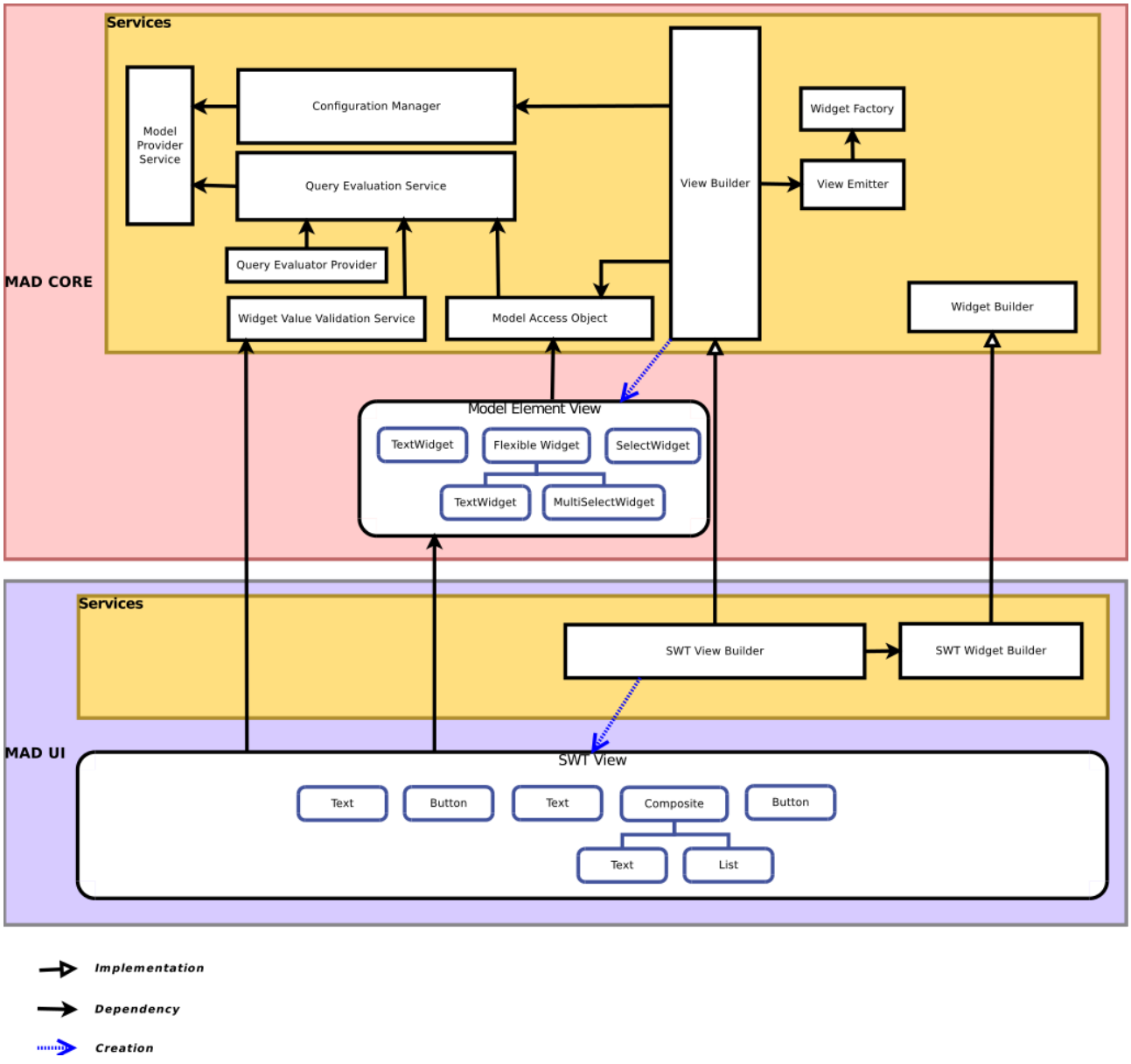
Toute modification d'une valeur par l'intermédiaire d'un widget doit être appliquée immédiatement à la propriété correspondante du modèle.

MAD manipule deux catégories de modèles, les modèles principaux : ceux déjà gérés par un éditeur de modèle et ceux dits étrangers qui présentent une relation avec le modèle principal. Les modèles étrangers sont ouverts par MAD pour permettre leur édition conjointement à un modèle principal. Les modifications des propriétés des modèles sont réalisées dans une transaction.

Les propriétés du **modèle principal** modifiées par MAD sont seulement mises à jour en mémoire et la sauvegarde du modèle est à la charge de son éditeur. La modification de propriétés appartenant à

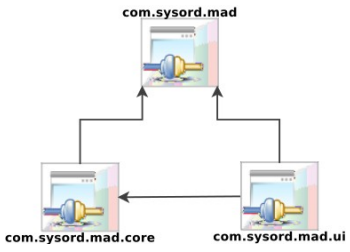
des **modèles étrangers** implique une sauvegarde de ces modèles par MAD.

Architecture :



MAD est réparti en trois plugins Eclipse principaux :

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



MAD est basé sur une architecture orientée services. Les interfaces de chaque service sont définies dans le plug-in *API* et leurs implémentations par défaut se situent dans le plugin *Core*. Il est possible de remplacer ou d'ajouter des services via le module Guice mis à disposition par le point d'extension défini dans le *Core*.

Services principaux :

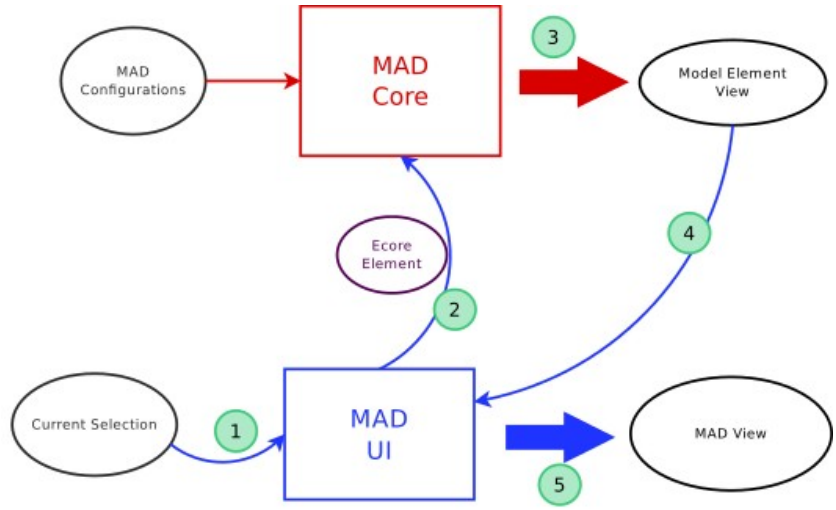
ConfigurationManager : Service de gestion des configurations MAD.
ModelProviderService : Service permettant d'obtenir des modèles EMF.
ModelAccessObject : Service d'accès et de persistance d'éléments dans des modèles EMF.
QueryEvaluationService : Service d'évaluation de requêtes sur les modèles EMF.
WidgetFactory : Service de création de widgets.
ViewBuilder : Service de création de la vue.
WidgetBuilder : Service de création de widgets spécifiques.
WidgetValueValidationService : Service de validation des valeurs des widgets.

L'*UI* utilise le *Core* de MAD pour créer une **vue Eclipse** représentant un formulaire d'édition en fonction de la configuration et de l'élément *Ecore* sélectionné dans l'environnement Eclipse.

Services redéfinis par l'*UI* :

SWTViewBuilder : Implémente le **ViewBuilder** pour générer la vue en SWT.
SWTWidgetBuilder : Implémente le **WidgetBuilder** pour générer les widgets spécifiques en SWT.

Cycle de vie :



View Generation for an Ecore element

Choix pour l'implémentation :

Mad sera évidemment un plugin Eclipse.

Configuration :

La configuration des formulaires d'édition MAD sera définie dans un modèle dédié. Étant donné la multitude d'informations que peut comporter une configuration et la lisibilité attendue de celle-ci nous avons opté pour un DSL textuel créé avec Xtext.

Xtext permet à partir d'une description de la grammaire d'un langage de générer automatiquement le

parseur et un modèle Ecore représentant les éléments du langage et leurs relations. Il produit également un éditeur totalement intégré à Eclipse qui fournit la complétion et la validation syntaxique. Un grand nombre de points d'extension ont été prévus pour ajouter à l'éditeur des modes de proposition personnalisés, des règles de validation spécifiques, une stratégie de formatage et bien d'autres choses. Il est également possible de personnaliser l'étape texte vers modèle.

Nous utilisons Xtext depuis plusieurs années et nous apprécions la facilité qu'il nous procure à créer rapidement des DSL textuels même si la mise au point et l'intégration du DSL à son environnement d'utilisation demeure souvent un projet dans le projet.

MAD permet d'éditer tout modèle Ecore, sa configuration étant un modèle Ecore, il serait possible d'utiliser MAD comme éditeur de sa propre configuration.

Intégration à Eclipse :

L'aspect graphique de MAD est une Vue Eclipse dont les widgets sont construits à la volée à partir de la configuration associée à l'élément sélectionné par un éditeur de modèle. Les widgets sont donc des widgets SWT construits avec l'aide de FormToolkit.

MAD peut toutefois créer des widgets autres que SWT, il suffit pour cela de fournir une implémentation personnalisée des services chargés de créer le container graphique de la vue et d' y ajouter les widgets (**ViewBuilder** et **WidgetBuilder**).

Évaluateurs de requêtes :

Les valeurs des widgets sont obtenues par l'évaluation de requêtes à partir de l'élément sélectionné. Nous avons opté pour **OCL** comme langage principal mais prévu la possibilité de créer et utiliser des évaluateurs personnalisés.

Les évaluateurs :

Pour son intégration à Eclipse, sa compatibilité avec Ecore, et son ouverture à l'extension, OCL nous a paru être un bon choix pour effectuer des requêtes sur le modèle.

OCL ne fournit de base que le moyen de lire dans un modèle. Lorsqu'il est nécessaire de modifier des propriétés, de créer ou supprimer des éléments il est possible d'invoquer à partir d'OCL des **EOperation** dédiées définies sur le métamodèle. UML met à disposition un grand nombre de ces opérations mais tous les modèles ne sont pas aussi généreux. Pour réaliser ce type d'opérations nous avons ajouté un évaluateur personnalisé nommé **MAD_EVALUATOR**. Il met à disposition un ensemble de fonctions simples permettant de créer, supprimer et déplacer des éléments du modèle.

Un évaluateur de requêtes **Acceleo** pour MAD est également disponible. Il permet de référencer et d'utiliser des requêtes définies dans des modules (.emtl) importés dans la configuration MAD.

Nous avions aussi intégré **EMF Query 2** mais étant donné l'état actuel d'avancement de ce projet, les apports de ce langage n'étaient pas assez intéressants et nous n'avons pas conservé cet évaluateur.

Espionner les évaluateurs :

Les valeurs à présenter sont fournies aux widgets à partir de requêtes de lecture. Toute valeur affectée à un widget éditable peut être modifiée par l'utilisateur et il faudra dans ce cas affecter la nouvelle valeur à la propriété correspondante dans le modèle. Il faut par conséquent connaître la propriété ayant fourni la valeur résultat de la requête. Nous avons pour cela utilisé deux méthodes pour « **espionner les évaluateurs »** et déterminer l'origine des résultats qu'ils fournissent.

La première spécialisée pour OCL consiste à intercepter les méthodes d'accès aux propriétés

et de collecter les informations contextuelles à cette lecture. Il est ainsi possible d'obtenir la propriété accédée et son élément propriétaire. En fin d'évaluation une validation de ces informations est effectuée par comparaison du résultat collecté et du résultat de l'évaluation. En cas de différences, l'**analyse de l'évaluation** est considérée comme invalide. Cette solution fournit un bon taux de réussite pour les requêtes simples mais se révèle totalement inefficace pour des requêtes utilisant les opérations sur les éléments du modèle ou les fonctions évoluées d'OCL.

La seconde solution se veut généralisable à tout évaluateur, elle consiste à instrumenter l'élément de base de la requête : **le contexte**, par le mécanisme des **proxy dynamiques**. Ce proxy intercepte les opérations invoquées sur l'élément pour collecter les accès à ses propriétés, lorsque le type du résultat le permet, un proxy avec le même comportement est retourné. En fin d'évaluation une validation de l'**analyse de l'évaluation** est effectuée. Cette méthode vient combler une partie des lacunes de la solution précédente mais comporte encore quelques limitations :

- Certains types non instrumentables jouent un rôle important dans l'évaluation mais échappent à la surveillance.
- Les traitements internes aux évaluateurs peuvent créer des structures non monitorées ou obtenir des résultats erronés en comparant les proxy.

Nous travaillons actuellement à des améliorations.

Lorsque l'**analyse de l'évaluation de la requête a échouée**, MAD ne peut pas déterminer l'action à réaliser pour effectuer la mise à jour de la valeur. Cette action peut alors être fournie explicitement dans la configuration en associant au widget la requête adaptée en **UPDATE_COMMAND**.

Synchronisation Vues modèle :

La synchronisation des vues avec le modèle consiste à garantir que toute modification effectuée sur un élément du modèle par MAD soit immédiatement prise en compte par la vue de l'éditeur du modèle. De la même manière, toute modification effectuée par l'éditeur du modèle, doit être répercutée sur la vue MAD.

La synchronisation de la vue de l'éditeur du modèle est à la charge de celui-ci et n'est pas de la responsabilité de MAD. Pour garantir que la vue MAD reflète l'image du modèle à tout instant, nous avons utilisé les **Adapters** fournis par EMF. Des Adapters placés sur l'élément édité notifient MAD de tous les changements, les widgets correspondant de la vue sont alors actualisés.

Injection de dépendances :

L'architecture de MAD se base sur un ensemble de services, chacun dédié à assurer une partie du flux de traitement nécessaire à constituer et gérer la vue pour un élément à éditer. Pour lier les services exploités par MAD nous avons eu recours à l'**injection de dépendances** qui assure un couplage lâche entre composants. Chaque service se présente sous la forme d'une interface, il peut exister une ou plusieurs implémentations pour chaque service. Une configuration définit pour tous ces services quelle implémentation doit être utilisée à l'exécution. Les utilisateurs de MAD peuvent fournir leur propre implémentation pour chaque service.

La version initiale de MAD a été développée pour **Eclipse Indigo**, pour cette raison nous n'avions pas la possibilité d'utiliser le mécanisme d'injection de dépendance d'**Eclipse 4** et avons opté pour **Guice**.

Requêtes:

Evaluateur OCL :

L'évaluateur OCL utilise la syntaxe standard. Pour faciliter l'écriture de certaines requêtes nous avons ajouté quelques opérations applicables sur tous les éléments.

`toString(context:OclAny):String`

Retourne un élément de type String à partir du contexte. Son implémentation consiste simplement à l'invocation de la méthode toString() Java sur l'objet contexte.

`rootContainer(context : OclAny) : EObject`

Si le contexte est un EObject l'élément racine de son modèle propriétaire est retourné. Dans tous les autres cas null est retourné.

`eContainer(context : OclAny) : EObject`

Si le contexte est un EObject son container est retourné. Dans tous les autres cas null est retourné.

Evaluateur MAD :

L'évaluateur MAD fournit quelques fonctions pour manipuler les éléments du modèle.

`CREATE(context:EObject, featureName:String, eClassName:String, container:EObject)`

Permet de créer un nouvel élément et de l'affecter à son container. Le nouvel élément créé est retourné.

Paramètres :

context : *le contexte. si le paramètre **container** n'est pas fourni, le context est considéré comme le container de l'élément à créer.*

featureName : *le nom de la feature du container devant contenir l'élément à créer*

eClassName : *le nom de la EClass de l'élément à créer.*

container : *(optionnel) container de l'élément à créer.*

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

Permet de modifier la valeur d'une propriété d'un élément.

Paramètres :

context : *le contexte, l'élément dont la propriété doit être modifiée.*

featureName : *le nom de la feature à modifier*

value: *la valeur à affecter.*

`DELETE(context:EObject)`

Permet de supprimer un élément de son modèle. L'élément et tous ses enfants sont supprimés, retirés de leur container et toutes leurs références sont supprimées.

La suppression est effectuée par l'appel de la méthode **EcoreUtil.delete(context, true)**

REMOVE(context:EObject)

Permet de retirer un élément de son container.

L'opération est effectuée par l'appel de la méthode **EcoreUtil.remove(context)**

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

Permet de déplacer vers le haut ou le bas un élément contenu dans une feature de type collection.

Evaluateur MAD EXTENSION:

L'évaluateur MAD Extension fournit quelques fonctions pour simplifier l'obtention et la synchronisation des éléments de modèles étrangers.

Chaînes de requêtes et sous-requêtes :

Les requêtes MAD peuvent être constituées de plusieurs requêtes écrites chacune dans un langage différent. Ce type d'écriture permet de combiner les fonctions des différents évaluateurs pour tirer le meilleur de chacun d'eux et faciliter la réalisation de requêtes complexes.

Requête élémentaire :

Une requête MAD simple comporte le corps de la requête sous la forme d'une chaîne ou une référence à une requête externe. L'identifiant du langage doit être fourni si la requête n'est pas en OCL.

```
"authors"
```

Pour un livre, retourne la liste des ses auteurs. Aucun langage n'est défini, OCL est utilisé par défaut.

```
language:ACCELEO call authorsOfSeveralBooks()
```

Appel d'un Query Acceleo qui pour un livre retourne la liste des ses auteurs étant aussi l'auteur d'autres livres de la bibliothèque.

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

Appel d'une fonction MAD pour un créer un nouvel élément livre à partir de la Library.

Chaîne de requêtes :

La requête MAD est constituée d'une liste de requêtes élémentaires ordonnées. Les requêtes sont évaluées les unes après les autres, le résultat d'une requête devient le contexte pour l'évaluation de la suivante.

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

Sélection du premier auteur du dernier livre de la bibliothèque ayant écrit plusieurs livres présents dans la bibliothèque.

Sous-requêtes :

Une requête principale peut comporter des sous-requêtes notées entre crochets. L'évaluateur commence par évaluer les sous-requêtes à partir du contexte, les résultats obtenus sont injectés dans la requête principale en remplacement de la sous-requête.

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

Pour un livre de la bibliothèque, retourne vrai si il est celui qui a le plus de pages. Le premier 'pages' s'applique aux livres de l'itération, '[pages]' évalué avant l'itération correspond au nombre de page du livre.

Variables contextuelles :

Dans le but de simplifier l'écriture des requêtes et de les rendre plus explicites, un ensemble de variables prédéfinies sont disponibles. Ces variables initialisées par MAD avant l'évaluation permettent d'accéder au contexte de la vue, à la valeur de l'élément édité et à bien d'autres informations contextuelles.

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

Règle de validation de la valeur d'un widget numérique. $UIVALUE est substitué par la valeur saisie dans le widget.

Evaluateurs personnalisés :

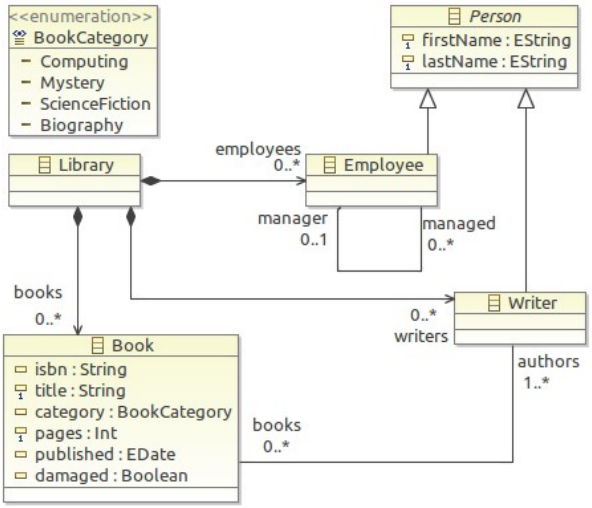
Pour satisfaire des besoins particuliers, MAD permet de créer ses propres évaluateurs, de les référencer dans la configuration et de les utiliser pour l'évaluation de requêtes. Un parseur d'expressions préfixées est disponible et peut être facilement réutilisé par tout nouveau langage.

Un évaluateur personnalisé est créé en réalisant l'interface **QueryEvaluator**.

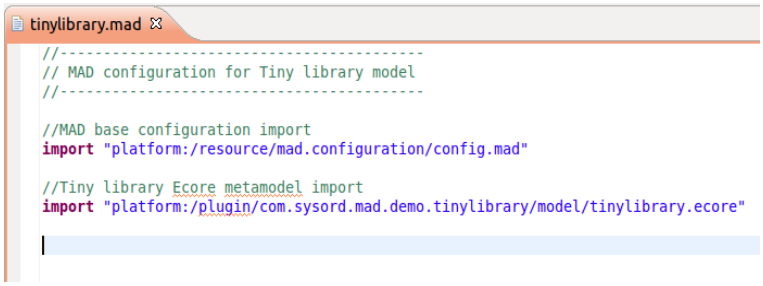
MAD en action :

La première étape pour utiliser MAD consiste à créer une configuration dans laquelle on définit pour le modèle à éditer les vues d'éditions que l'on souhaite obtenir. Il suffit ensuite d'ajouter cette configuration aux préférences MAD et c'est tout, l'éditeur de propriétés est prêt. Toutes les modifications de la configuration seront prises en compte immédiatement.

Les exemples qui vont suivre utiliseront le modèle « tinylibrary » qui est une adaptation simplifiée du modèle Library.



Création d'une configuration MAD pour les modèles ***tinylibrary***. A ce point, aucune vue pour l'édition n'est définie.



Widgets :

Mad fournit une palette de widgets configurables pour la constitution de formulaires personnalisés.

Widgets pour l'édition:

- Saisie de valeurs de type Texte, Nombre, Date et Booléen.
- Sélection de valeurs: Combobox, Liste et PickList embarquées ou en popup.
- Editeur d'éléments de modèle Xtext (utilisation de l'éditeur embarqué de Xtext).

Widgets pour la visualisation: Affichage en lecture seule d'une valeur élémentaire ou calculée.

Widget de navigation: Liste d'éléments proposés permettant d'accéder directement à leur vue détaillée.

Widget flexible : Affichage d'une liste d'éléments éditables, chaque élément est représenté par l'ensemble des widgets décrits dans son template associé.

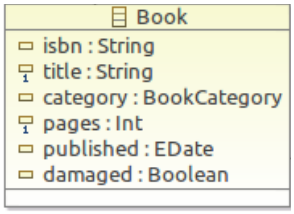
Widget composite: Inclusion dans un template de tous les widgets d'un template déjà défini pour représenter un élément composite dont on souhaite homogénéiser le rendu de chacune de ses occurrences.

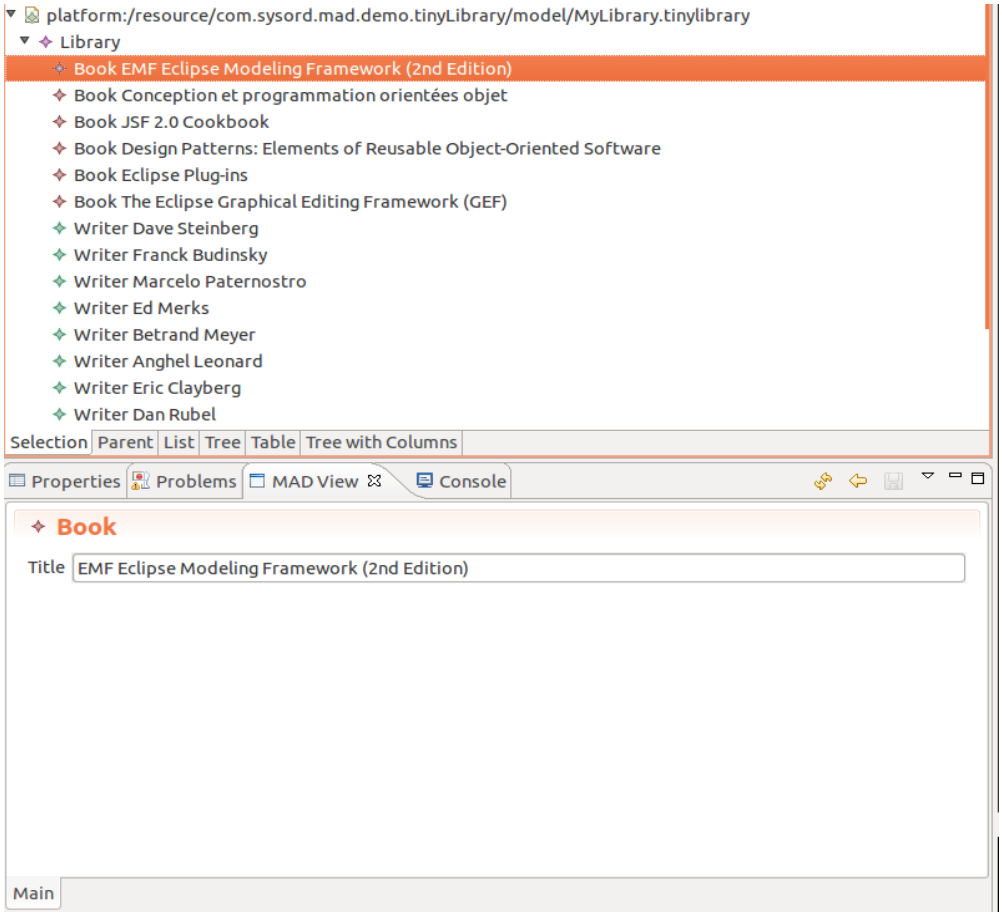
Commande : bouton pour le lancement d'une action.

La palette de widgets fournie par MAD est totalement extensible, il est possible de spécialiser chacun des widgets d'origines en créant ses propres implémentations.

Une première configuration :

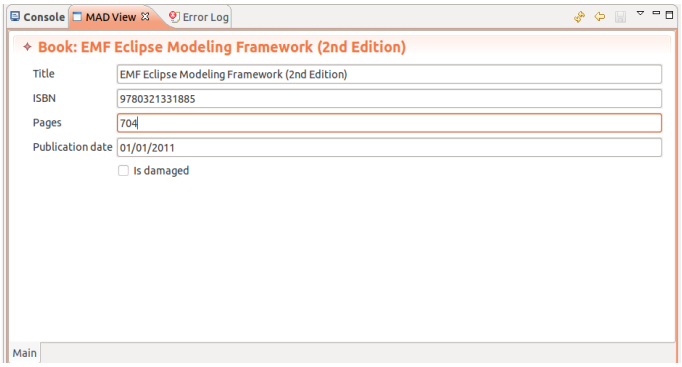
```
//-----------------------------------------
// MAD configuration for Tiny library model
//-----------------------------------------

//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.
}
```





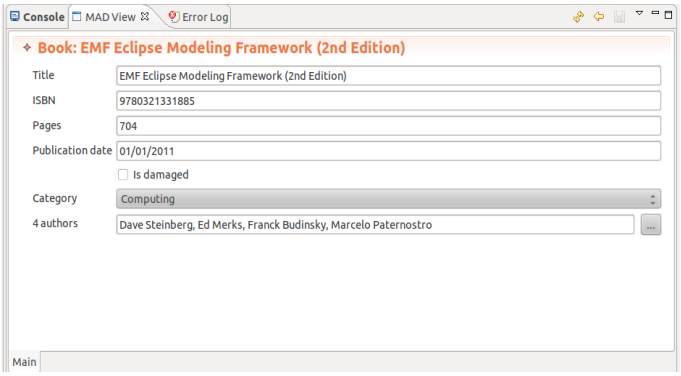
Widgets simples :

```
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"

}
```



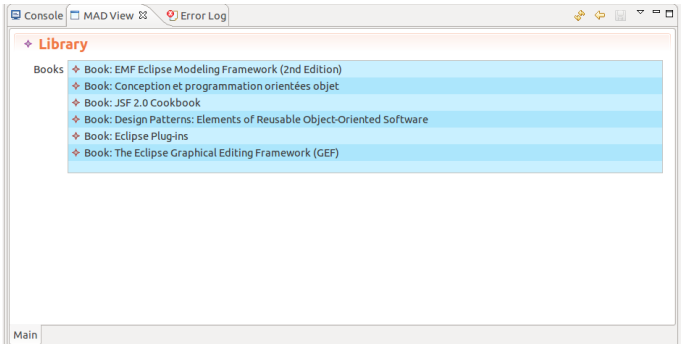
Widgets Listes :

```
     ...

    //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]"
```



Widget pour la navigation :

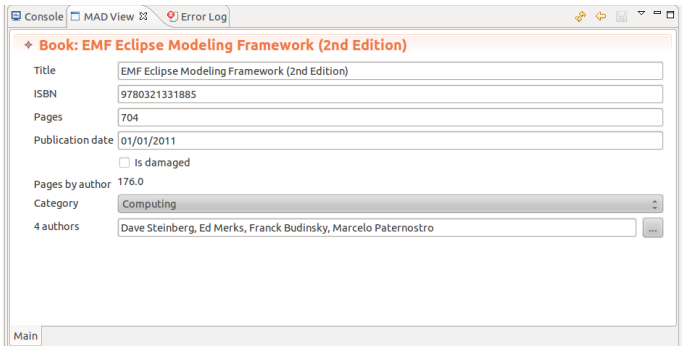
```
//Configuration for the Library element
Configuration LIBRARY for tinylibrary.Library {

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



Le widget de navigation propose une liste d'éléments. Un double-clic sur un élément permet de naviguer vers sa vue détaillée. La flèche vers la gauche permet de revenir à la vue précédente.

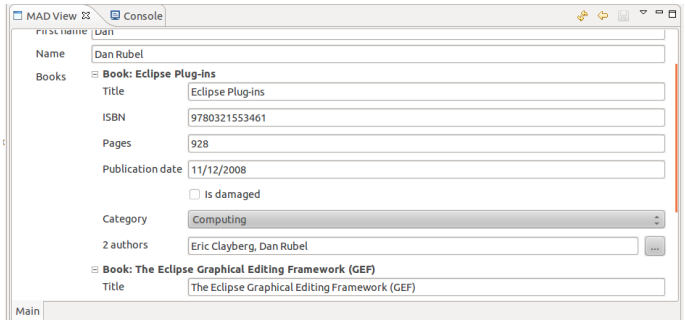
Widget pour affichage en lecture seule:

```
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
}
```



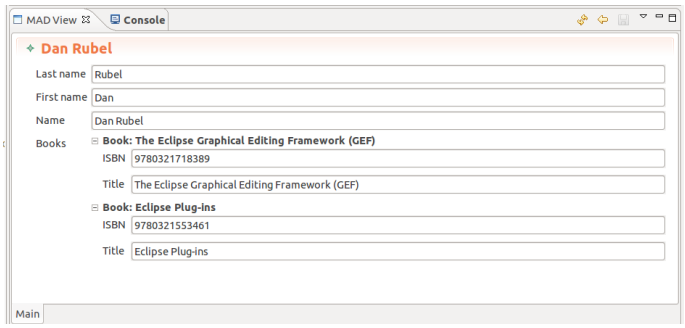
Flexible widget :

```
//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"
}
```



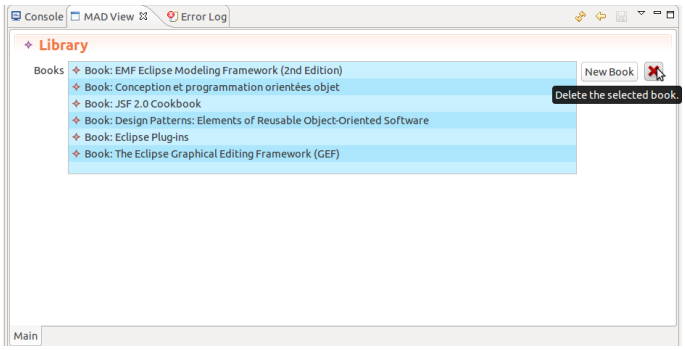
Sélection d'un template particulier pour les éléments du flexible :

```
//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"
}
```



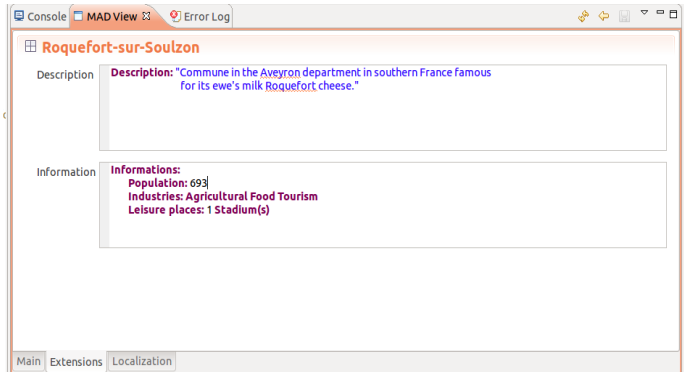
Widget commande :

```
//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.")
}
```



Autres widgets :

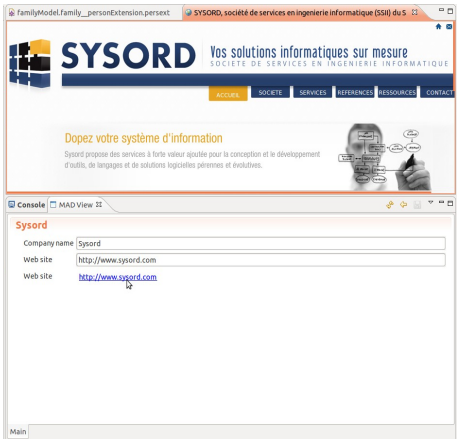
Editeur Xtext :

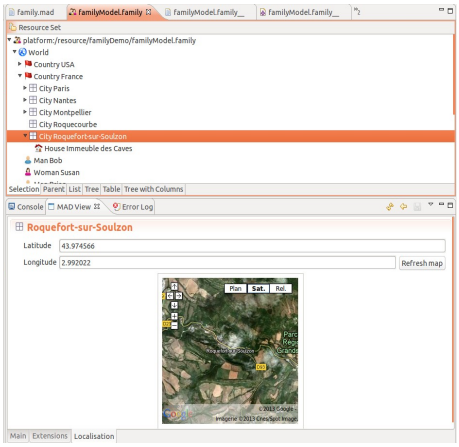


Le widget éditeur **Xtext** permet d'intégrer à la vue MAD des éditeurs embarqués pour modifier des éléments de modèles Xtext en profitant de la complétion et la validation.

La réalisation de ce widget nous a posé quelques problèmes liés au fonctionnement de Xtext. Toute modification dans le texte déclenche un appel au parseur et la reconstruction de toute la branche du modèle impactée. Pour cette raison, garder des références ou placer des Adapters sur des éléments de modèle Xtext dont les sous-éléments peuvent changer est inutile puisque ces références peuvent devenir obsolètes. La modification de l'un ou l'autre des textes de l'exemple ci-dessus provoque l'éviction du modèle de l'autre. Une copie en mémoire de l'élément édité et de son URI en prévision d'une fusion sur modification paraît un bon compromis. Cependant si l'URI de l'élément n'est pas basée sur un identifiant ou s'il doit exister des relations (références, règles de validation) entre les deux éléments, l'édition ou la fusion ne pourront s'effectuer correctement. Il est donc nécessaire de recharger tous les widget Xtext potentiellement impactés de la vue lorsque l'un d'eux applique une modification sur le modèle.

Html Link et Google MAP widgets :

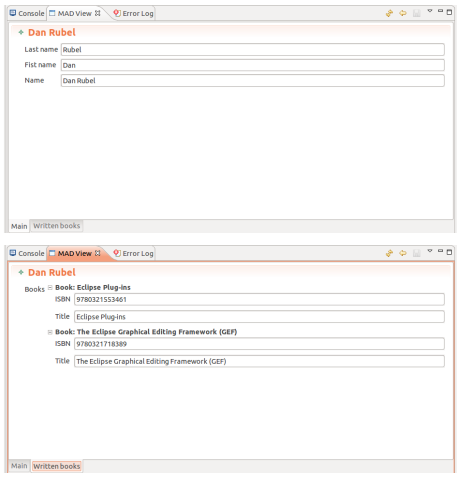




Ces widgets ont été conçus en personnalisant le **`OUTPUTTEXT_WIDGET`**. Leur configuration détaillée est présentée dans une vidéo de la première version de MAD : Mad is Customizable

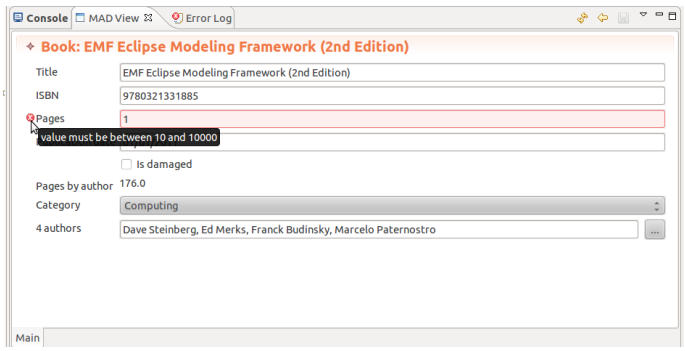
Layout :

```
//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
}
```



Validateurs :

```
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]"
    ...
}
```

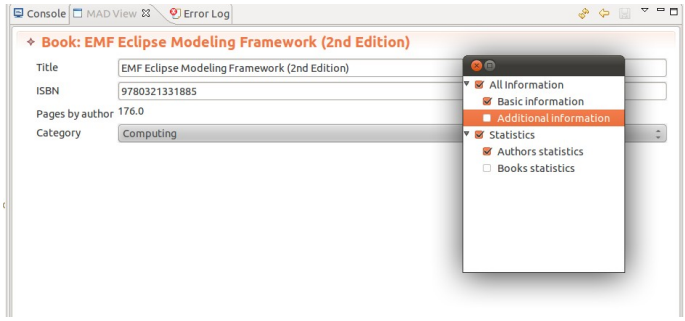


Des règles de validations placées sur les widgets permettent de vérifier la conformité des valeurs avant la mise à jour du modèle.

Layers :

Les layers permettent de rassembler et de sélectionner les widgets à afficher en fonction d'une thématique particulière. Leur intérêt est de rassembler les widgets présentant des informations d'un même domaine. Ces couches sont superposables et organisées hiérarchiquement. Elles permettent d'affiner une vue selon l'intérêt de l'utilisateur.

```
//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 supporte l'internationalisation, en s'appuyant sur les fonctionnalités NLS (National Language Support) d'Eclipse. Les traductions sont stockées dans des fichiers de propriétés par locale, un nom symbolique est associé à une traduction dans la locale cible. Dans la configuration MAD les textes élémentaires ou calculés à traduire sont précédés du **mot-clé I18N.** Leur valeur contient le nom symbolique de la traduction à fournir suivi optionnellement de requêtes destinées à calculer et fournir les parties variables (paramètres) de la traduction. MAD effectue les traductions vers la Locale par défaut du système. Tous les labels d'une configuration MAD sont internationalisables.

Intérêts de MAD et cas d'utilisation:

MAD permet l'édition personnalisée de tout modèle Ecore, il est par conséquent un outil utilisable dans toutes les approches MDE et MDA mises en application dans l'environnement Eclipse. La création et la maintenance de modèles est une tâche très importante qui doit s'effectuer avec un maximum de confort pour l'utilisateur. En fournissant aux utilisateurs un éditeur adapté à leurs besoins ou un complément aux modeleurs et éditeurs qu'ils utilisent, MAD améliore l'expérience utilisateur.

Par son aspect dynamique, MAD permet la création rapide d'une interface pour un utilisateur en suivant ses directives. Ce mode de conception comporte un grand nombre de similitudes avec l'approche RAD (Rapid Application Developement) dans laquelle un utilisateur décrit ses besoins, le concepteur configure un générateur et le résultat est immédiatement évalué par l'utilisateur. Au fil d'itérations rapides l'utilisateur obtient le produit attendu. Avec MAD la phase de génération n'existe pas et toute modification de la configuration est immédiatement visible. L'utilisateur décrit son interface idéale. Le concepteur doit uniquement avoir des notions sur la manière d'écrire les requêtes et disposer d'une bonne connaissance du modèle cible. Il configure MAD en fonction de la description des besoins de l'utilisateur, l'utilisateur voit immédiatement le rendu, livre ses impressions et valide le résultat ou demande des améliorations.

Un autre intérêt de MAD est d'assurer un mode d'édition identique pour un même élément quel que soit le modeleur ou l'éditeur de modèles. Ainsi l'édition d'un élément UML avec l'éditeur de modèle UML d'Eclipse, Papyrus ou UML Designer comportera un maximum de similitudes permettant à un utilisateur de retrouver ses habitudes et de réduire les temps d'adaptation nécessaires à la prise en main d'un nouveau modeleur.

MAD permet l'édition conjointe d'éléments provenant de plusieurs modèles, il est de ce fait tout à fait adapté pour la décoration ou l'extension de modèles. Un modèle principal qui comporte les informations principales du domaine est édité, un modèle complémentaire comporte des informations additionnelles ciblées dans le cadre d'une utilisation spécifique. MAD permet de fusionner ces informations sur une même vue d'édition et de modifier ou consulter conjointement les deux modèles. L'utilisation des « Layer » permet d'affiner la vue issue de cette fusion.

Dans le cas d'une décoration, le plus souvent il correspond à chaque élément du modèle principal décoré un homologue dans le modèle décorateur. Un des problèmes sous-jacent de ce type d'édition est la nécessité de synchroniser les deux modèles : toute création d'un élément décorable dans le modèle principal nécessite la création de son homologue dans le modèle décorateur. MAD met à disposition des services dédiés pour faciliter l'extension et la décoration de modèles.

ModelExtensionManager : est un service utilisé par le langage de requête MAD_EXTENSION. MAD_EXTENSION permet de formuler des requêtes pour retrouver dans un modèle d'extension ou de décoration l'élément correspondant à un élément du modèle principal. Il suffit de fournir une implémentation de l'interface **ModelExtensionManager** qui réalise la stratégie de correspondance à adopter pour faire le lien entre un modèle principal et un modèle d'extension ou de décoration.

ExtensionModelSynchronizer: est un service utilisé pour synchroniser le modèle principal avec un modèle d'extension. C'est un observateur du modèle principal qui surveille toutes les créations et suppressions d'éléments et utilise le **ModelExtensionManager** pour répercuter ces modifications dans le modèle d'extension. Les implémentations à utiliser pour un modèle sont définies dans la configuration de MAD.

(démos en vidéos :Family Model Extension, MAD avec Xtext)

Enfin MAD est évolutif et ouvert, il permet facilement de s'adapter à de nouveaux besoins dès leur identification par configuration ou par extension : intégration ou développement de nouveaux langages de requête spécifiques, création de widgets et de générateurs de vues personnalisés pour un nouvel environnement graphique, réutilisation du cœur de MAD dans des applications.

Conclusion :

MAD est aujourd'hui fonctionnel, nous l'utilisons en interne chez Sysord pour la conception et l'édition de tous nos modèles et plus particulièrement pour les modèles UML utilisés en entrée de nos générateurs d'applications (vidéo de démo : MAD avec UML) . Un certain nombre d'évolutions sont en cours pour améliorer les performances en incluant un compilateur de requêtes.

Nous envisageons également un certain nombre d'évolutions qui sont en attente de réalisation :

- Environnement de configuration fournissant des assistants pour la création et la validation des requêtes.
- Intégration de CDO pour gérer le mode multi-utilisateur et les verrouillages.
- Générateur de configuration MAD : à partir d'un élément sélectionné pour lequel aucun template n'est défini, un bouton permet de produire par introspection de l'élément une description par défaut et de l'ajouter automatiquement à la configuration en cours.
- Possibilité de créer des commandes invoquant des méthodes Java par reflection.
- Générateur d'éditeur MAD : à partir d'une configuration éprouvée en utilisation interprétée, génération d'un plugin Eclipse.
- Edition d'éléments multimédia : son, image, vidéo, charts et graphs.
- Exploitation des avantages de Eclipse 4 et pourquoi pas JavaFX.