

**REPRÉSENTATION DES
CONNAISSANCES : FRAME
REPRESENTATION LANGUAGE**

par Albéric Martel et Romain Bouleis

TABLE DE MATIÈRES

TABLE DE MATIÈRES.....	2
ÉTAT DE L'ART.....	3
SOLUTION ADOPTÉE.....	4
Lancement de l'application.....	5
Chargement et sauvegarde d'un schéma.....	5
Creation et suppression d'une frame.....	5
Visualisation de la relation ako.....	6
Creation et suppression d'un slot.....	6
Modification des facettes d'un slot.....	7
Le Frame-Context.....	7
Utilisation de variables dans les attachements procéduraux.....	8
CONCEPTION DE L'APPLICATION.....	9
Les fichiers BOSS.....	9
La classe FrameList.....	10
La classe FrameContext.....	10
La classe Frame.....	10
La classe Slot.....	13
CONCLUSION.....	17

ÉTAT DE L'ART

Il existe différents moyens de représenter les connaissances ; les systèmes à base de schémas sont l'un de ces moyens. Ils tiennent leur origine de l'intelligence artificielle et de la psychologie. Ces systèmes sont nécessaires car les « éléments » de la connaissance doivent être regroupés et structurés.

Parmi ces systèmes, on retrouve le « frame representation language ». Il a été développé par Marvin Minsky au Massachusetts Institute of Technology avec le concept de frame et le « Frame Representation Language ». Ce dernier est constitué de plusieurs éléments :

- Des frames : Ce sont des structures de données complexes qui représentent des concepts. Elles ont un nom et une série d'attributs appelés des slots.
- Des slots : Ce sont des propriétés de la frame, ils définissent la structure de données. Par exemple, un concept peut nécessiter d'avoir un type, une durée, on utilisera les slots pour les représenter.
- Des facettes : Chaque slot comporte ce qu'on appelle des facettes, qui sont au nombre de 3. La première est la valeur du slot, c'est une facette déclarative. Les deux autres sont des facettes procédurales. Elles comportent du code à exécuter appelé « attachement procédural ». Elles sont activées à chaque accès à la valeur du slot. La première, « if-added » est exécutée à chaque ajout de valeur à la facette « value », la seconde, « if-needed » lorsque la facette « value » n'a pas de valeur.
- Des relations : Il existe des relations entre les frames. Elles sont représentées par des slots. Une relation particulière est la relation ako, autrement dit la relation d'héritage, elle permet l'accès aux attributs de la frame mère. Des exemples de relations non héritées sont « instance », « possède », etc ...

Le frame representation language permet donc une programmation dirigée par les données. Les appels se font par effets de bord.

SOLUTION ADOPTÉE

Dans le cadre de travaux pratiques de « Représentation des connaissances », nous avons eu à implémenter un langage de représentation de frames et une interface de saisie et visualisation associée. Pour se faire, nous avons choisi Smalltalk comme langage de programmation car il permet l'exécution de code « à la volée », ce qui est nécessaire pour avoir des attachements procéduraux dynamique. De plus Smalltalk à de nombreux avantages qui seront détaillés par la suite.

Avant de poursuivre, voici une copie d'écran de l'application :

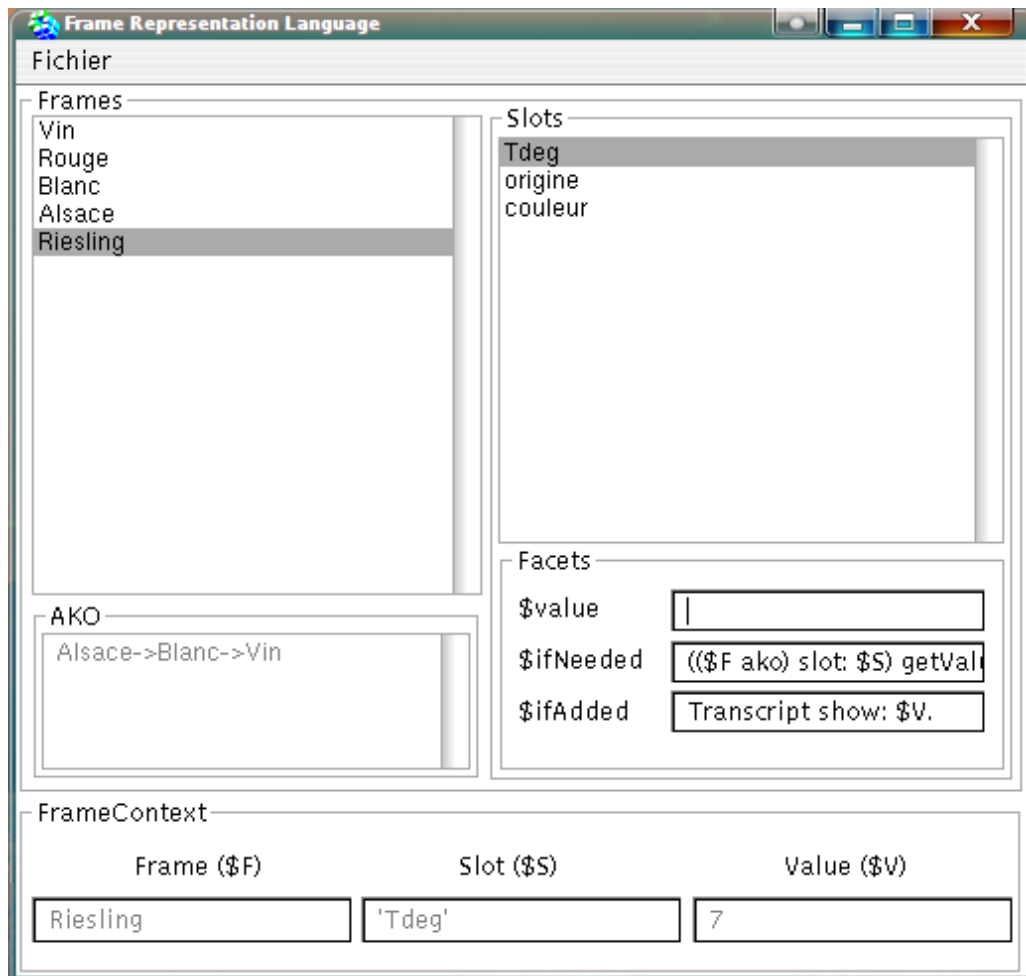


Illustration 1: Application

La fenêtre est divisée en deux parties majeures, elles même divisées en sous parties :

- Les frames

Ce cadre contient la liste des frames ainsi que :

- La relation AKO de la frame sélectionnée,
- La liste des slots associés à la frame sélectionnée,
- La liste des facettes du slot sélectionné, et leurs valeurs.

- Le « FrameContext »

Il contient les éléments courants, à savoir la frame actuelle, le slot actuel et la valeur du slot actuel.

LANCEMENT DE L'APPLICATION

Après avoir chargé le programme dans VisualWorks (version 7.5 et supérieurs), le programme pourra être lancé via l'exécution de la ligne suivante dans un workspace :

FrameViewer openWithModel: (FrameList new) withContext: FrameContext new.

CHARGEMENT ET SAUVEGARDE D'UN SCHÉMA

L'écriture d'un schéma ne se fait généralement pas « d'une traite », tout comme un programme ou un document texte, on est souvent amené à le modifier, à le compléter. Il nous est donc apparu indispensable de pouvoir sauvegarder et charger un schéma. Ceci se fait grâce au menu « Fichier » :

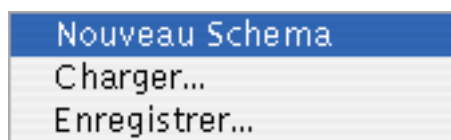


Illustration 2: Menu fichier

Il y est donc possible de :

- Charger un schéma depuis un fichier existant,
- Enregistrer un schéma dans un fichier,
- Effacer le schéma courant et en commencer un nouveau.

CREATION ET SUPPRESSION D'UNE FRAME

La création et la suppression d'une frame se font grâce à un clic droit dans la liste des frames. Un menu apparaîtra permettant donc d'ajouter une frame ou de supprimer la frame sélectionnée. Si aucune frame n'est sélectionnée, le menu de suppression sera inaccessible :

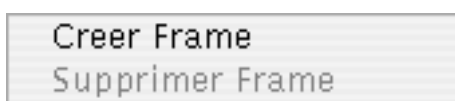


Illustration 3: Menu contextuel sur les frames

Il convient de différencier deux cas de création :

- Si une frame est sélectionnée, la frame nouvellement créée aura pour relation AKO la frame sélectionnée. C'est le seul moyen de définir la relation AKO.
- Si aucune frame n'est sélectionnée, la nouvelle frame n'aura pas de relation AKO.

Enfin, il faudra donner un nom à la nouvelle frame grâce à l'invite suivante :

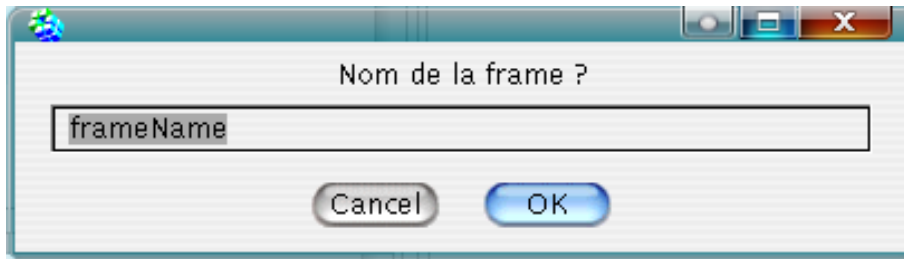


Illustration 4: Invite nom de la frame

Il est a noté que la suppression d'une frame ayant des frames « filles » aura pour incidence de remplacer la relation AKO des frames filles par la relation AKO de la frame à supprimer. Ceci vaut quelque soit la relation AKO : une autre frame ou *nil*. Par exemple, soit la frame *Riesling* dont l'ascendance est la suivante :

Alsace -> Blanc -> Vin

La suppression de la framme *Blanc* engendrera l'ascendance suivante pour la frame *Riesling* :

Alsace -> Vin

VISUALISATION DE LA RELATION AKO

La relation AKO d'une frame est visible dans le cadre « AKO » de l'application :



Illustration 5: Visualisation de la relation AKO

CREATION ET SUPPRESSION D'UN SLOT

De la même façon, un menu contextuel accessible via un clic droit sur la liste des slots permet d'en créer et d'en supprimer :

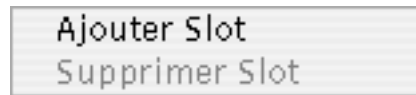


Illustration 6: Menu contextuel sur les slots

Ici aussi, il n'est possible d'ajouter un slot que si une frame est sélectionnée. Supprimer un slot sera accessible si un slot et une frame sont sélectionnés.

MODIFICATION DES FACETTES D'UN SLOT

Un slot possède trois facettes :

- \$value : il s'agit de la valeur du slot,
- \$if-needed : c'est un attachement procédurale. Autrement dit, un programme qui sera exécuté si la valeur du slot est requise mais que celle-ci est vide. La valeur résultant de l'exécution de cet attachement ne sera pas placée dans la facette \$value.
- \$if-added : il s'agit aussi d'un attachement procédurale. Il sera exécuté après l'ajout d'une valeur à \$value. Un cas typique d'utilisation est la gestion de logs.

Tous ces slots sont modifiables dans les champs prévus à cet effet :

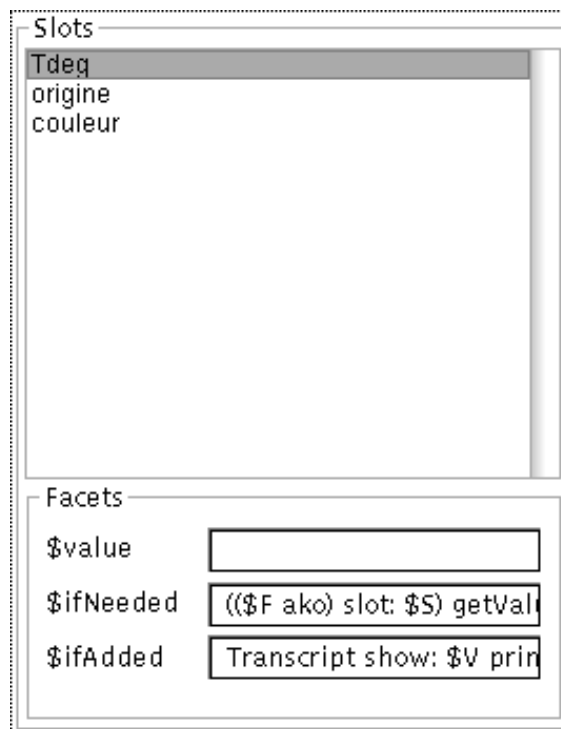


Illustration 7: Modification des facettes

LE FRAME-CONTEXT

Le frame-context contient trois valeurs non modifiables directement :

- La frame courante (sélectionnée),
- Le slot courant (sélectionné),

- La valeur du slot courant.

La valeur d'un slot est le résultat de l'appelle *getValue* sur ce slot. Nous avons implémenté *getValue* de la manière suivante :

- Si la facette *\$value* du slot n'est pas vide, renvoie la valeur de cette facette,
- Sinon, si la facette *\$if-needed* n'est pas vide, exécute cet attachement procédurale. Si une valeur est retournée suite à cette exécution, renvoie cette valeur,
- Sinon, parcourt la hiérarchie de la frame du slot à la recherche d'un slot de même nom. Si un slot est trouvé, applique les deux opérations précédentes sur ce slot. Si le slot trouvé ne renvoie pas de valeur via *\$value* ou *\$if-needed*, continue de parcourir la hiérarchie,
- Sinon, renvoi *nil*.

Voici le cadre contenant le frame-context :

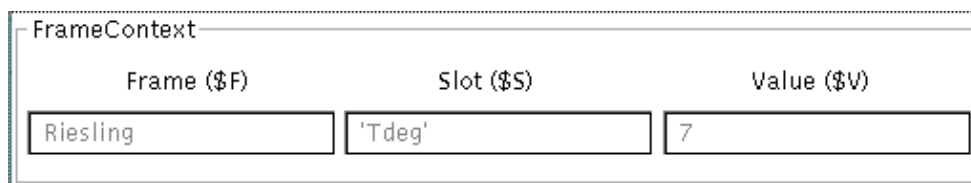


Illustration 8: Frame-Context

UTILISATION DE VARIABLES DANS LES ATTACHEMENTS PROCÉDURAUX

Pour plus de souplesse et pour décupler les possibilités, il est possible d'utiliser les trois valeurs du frame-context dans les attachements procéduraux *\$if-added* et *\$if-needed*. Ceci se fait grâce aux trois pseudos-variables *\$F*, *\$S* et *\$V* qui correspondent respectivement à la frame courante, au slot courant et à la valeur de la facette *\$value* du slot courant. Leurs utilisations nécessitent néanmoins un complément pour chacune d'elle :

- *\$F* est l'objet en tant que tel représentant la frame, on a ainsi accès à toutes ses méthodes (voir le chapitre suivant).
- *\$S*, au contraire est le nom du slot courant, et non un objet Smalltalk. Pour avoir accès à celui-ci, il faut y accéder via le slot courant de la manière suivante :

(\$F slot: \$S)

L'objet ainsi retourné représente le slot courant, on peut donc appeler toutes les méthodes de la classe Slot.

- *\$V* représente la valeur de la facette *\$value* du slot, et non la valeur que pourrait retourner *getValue*.

Il convient néanmoins d'utiliser ces variables avec précaution, en effet, utiliser

(\$F slot: \$S) getValue

à l'intérieur de l'attachement procédural *\$if-needed* entraînera une boucle infinie si il n'existe pas de valeur dans *\$value* (*getValue* faisant appel à *\$if-needed* dans ce cas là).

CONCEPTION DE L'APPLICATION

LES FICHIERS BOSS

Les fichiers boss sont les fichiers dans lesquels Smalltalk est capable de stocker des objets tels qu'ils sont en mémoire. C'est grâce à ce système que nous sauvegardons la liste de frames constituant un schéma. L'une de leur caractéristique est de n'être pas « lisibles » et donc modifiables par un homme comme peuvent l'être les fichiers XML. C'est pour cette raison que nous les avons choisis, nous ne voulions pas qu'un utilisateur puisse modifier un schéma hors de l'interface. En effet, il pourrait supprimer une frame sans faire attention à ses frames filles, par exemple.

Voici le procédures permettant de sauvegarder et charger une liste de frames :

```
saveFile
  | of myWriteStream chemin bos |
  of := Tools.SaveFileDialog new.
  of open.
  of cancel value iffFalse: [
    chemin := of selection asString.
    myWriteStream := chemin asFilename writeStream.
    bos := BinaryObjectStorage onNew: myWriteStream.
    [bos nextPutAll: (self model frames) ] ensure: [bos close].
  ]

loadFile
  | of chemin bos myReadStream frames allFrames |
  of := Tools.OpenFileDialog new.
  of open.
  chemin := of fileList selection.
  chemin isNil iffFalse: [
    myReadStream := chemin asFilename readStream.
    bos := BinaryObjectStorage onOldNoScan: myReadStream.
    [frames := bos contents] ensure: [bos close].
    allFrames := FrameList new.
    frames do: [:f |
      allFrames addFrame: f.
    ].
    self closeRequest.
    self = nil.
    FrameViewer openWithModel: allFrames withContext:
FrameContext new.
  ]
```

LA CLASSE FRAMELIST

Cette classe contient simplement une liste de frames. Elle est le modèle de l'interface. Elle possède trois méthodes :

- addFrame: f qui ajoute une frame à la liste,
- frames qui retourne la liste des frames,
- removeFrame: f qui supprime une frame de la liste et gère la relation AKO de ses classes filles. Voici son code :

```
removeFrame: f
    "supprime la frame de la liste"
    self frames remove: f.
    "recherche ses sous classes pour supprimer la relation ako"
    self frames do: [:fr |
        (fr ako = f) ifTrue: [
            fr removeInheritance.
            (f ako = nil) ifFalse: [
                fr inheritFrom: f ako.
            ]
        ]
    ].
```

LA CLASSE FRAMECONTEXT

Le frame-context est aussi représenté par un objet. Celui possède naturellement une Frame, un nom de Slot, et une valeur. L'accès au frame se fait grâce aux asseurs F et $F:f$, il en va de même pour l'accès au slot, avec S et $S:s$. Pour valeur, seul le « getter » est disponible via V . En effet, l'affectation de la valeur courante ne se fait que lors d'un changement de sélection du slot, autrement dit, lors de l'affectation du slot via $S:s$. Voici donc le code de l'affectation du slot :

```
S:s
    S := s.
    (S = nil) ifFalse: [V:= (F slot:S) getValue]
    ifTrue: [V:= nil].
    self changed: s.
```

Comme il a été cité précédemment, le programme est lancé avec comme second paramètre un objet FrameContext. Celui ci sera affecté à une variable globale, nommé *FrameCtxt*. Il sera ainsi possible d'accéder au frame-context depuis n'importe quelle partie du programme, et notamment les attachements procéduraux.

LA CLASSE FRAME

Une frame contient trois attributs :

- Un nom,
- Un dictionnaire de chaînes de caractères associées à des slots,

- Une frame AKO.

Une frame est construite avec un nom, si ce n'est pas le cas, un nom par défaut lui est attribuée. Pour parvenir à ceci, nous avons redéfini la méthode de class *new* et la méthode d'instance *initialize* :

```
new
  ^self new initialize: 'no names'

new: n
  ^(super new) initialize: n.

initialize: n
  slots := Dictionary new.
  name := n.
  ^self
```

De plus, la classe Frame possède les méthodes d'instances suivantes :

- Dans le protocole *printing* :

- printString

Cette méthode renvoie simplement le nom de la frame.

- printDetails

Renvoie une chaîne de caractère contenant des détails sur la frame. Par exemple :

slots : [Tdeg origine couleur]
ako : Alsace->Blanc->Vin

- printAKO

Renvoie une chaîne contenant la hiérarchie de la frame. Si la frame n'a pas d'ascendance, renvoie la chaîne *nil*. Exemple :

Alsace->Blanc->Vin

- Dans le protocole *accessing* :

- addSlot: nom

Ajoute un nouveau slot de nom *nom* à la frame. Voici la méthode :

```
addSlot: nom
  |s|
  s := Slot new.
  s frame: self.
  self slots at: nom put: s.
```

- ako

Renvoie la valeur de l'attribut *ako*.

- getAllSuperFrames

Retourne une liste contenant la hierarchie de la frame :

```
getAllSuperFrames
| |
l := List new.
self getAllSuperFrames: l.
^l.
```

- getAllSuperFrames: liste

Recherche les frames ascendantes et retourne une liste. Cette méthode est appelée uniquement par elle même et par getAllSuperFrames :

```
getAllSuperFrames: superFrames
(self ako = nil) ifTrue: [ ^superFrames. ]
ifFalse: [
    superFrames add: ako.
    ako getAllSuperFrames: superFrames.
].
```

- getAllSuperFramesAsString

Renvoie sous forme de chaîne la liste des frames constituant sa hierarchie. Equivalent à la méthode *printAKO*.

- hasSlot: s

Renvoie un booléen indiquant si la frame contient le slot passé en paramètre :

```
hasSlot: s
self slots at: s ifAbsent: [ ^false ].
^true.
```

- inheritFrom: frame

Permet de définir la relation d'héritage (AKO). Nous avons décider d'utiliser cette méthode plutôt que le traditionnelle « setter » *ako* : *frame* afin de pouvoir s'assurer que le paramètre est bien une frame :

```
inheritFrom: frame
(frame isKindOf: Frame) ifTrue: [ self ako:frame. ]
ifFalse: [ Dialog warn: ('Vous essayez d''heriter d''un objet qui n''est pas une
Frame !') withCRs ].
```

- name

Retourne le nom de la frame. Equivalent à *printString*.

- removeInheritance

Affecte la valeur *nil* à *ako* (permet de supprimer la relation d'héritage) :

```
removeInheritance
  ako := nil.
```

- removeSlot: s

Permet de supprimer un slot de la frame :

```
removeSlot: s
  self slots removeKey: s.
```

- slot: s

Renvoie le slot de nom *s*. Si il n'existe pas, le crée et renvoie le nouveau slot :

```
slot: nomDuSlot
  self slots at: nomDuSlot ifAbsent: [ self addSlot:nomDuSlot ].
  ^self slots at: nomDuSlot.
```

- slots

Renvoie le dictionnaire contenant les noms associés aux slots de la frame.

Ces méthodes sont directement accessibles depuis les attachements procéduraux grâce à la pseudo-variable \$F.

LA CLASSE SLOT

Nous avons défini une classe Slot. Un slot est constitué :

- De facets : c'est un dictionnaire, chaque facette sera accédée par une clé,
- Du frame auquel le slot appartient.

La classe possède les deux accesseurs pour ces attributs ainsi que les méthodes suivantes :

- Dans le protocole *printing* :
 - printString

Retourne la valeur du slot courant :

```
printString
  ^self getValue.
```

- printFacets

Retourne une chaîne de caractères contenant la liste des facettes associées à leur contenu. Par exemple :

```
'$if-needed : "self halt"
$if-added : "self halt"
$value : "Ma valeur" '
```

- Protocole private
 - replaceFrameContextSymbols: s

Lors de l'utilisation des pseudos variables \$F, \$S et \$V dans les attachements procéduraux du slot, il est nécessaire de convertir ces chaînes de caractères en code exécutable par Smalltalk. Ainsi, chaque variable sera remplacée par une autre chaîne faisant référence au frame-context. Les variables seront ainsi respectivement remplacées par :

- 'FrameCtxt F'
- 'FrameCtxt S'
- ((FrameCtxt F slot: FrameCtxt S) getFacetValue: "\$value")

Ces chaînes seront compréhensibles grâce à la variable globale *FrameCtxt* vue précédemment.

Voici cette méthode :

```
replaceFrameContextSymbols: s
  | str |
  str := s.
  str := str copyReplaceAll: '$F' with: 'FrameCtxt F'.
  str := str copyReplaceAll: '$S' with: 'FrameCtxt S'.
  str := str copyReplaceAll: '$V' with: '((FrameCtxt F slot: FrameCtxt S)
getFacetValue: "$value") '.
^str.
```

- Dans le protocole *controlling* :
 - executelfAdded

Cette méthode permet l'exécution du code renseigné dans la facette \$if-added du slot :

```
executelfAdded
  | ifAdded |
  ifAdded := self getFacetValue: '$if-added'.
  (ifAdded = nil) ifFalse: [
    ifAdded := self replaceFrameContextSymbols: ifAdded.
    Compiler evaluate: ifAdded.
  ].
```

- executelfNeeded

Méthode qui évalue le contenu de la facette \$if-needed :

```
executelfNeeded
  | needed |
  needed := self getFacetValue: '$if-needed'.
  (needed = nil) ifFalse: [
    needed := self replaceFrameContextSymbols: needed.
    ^Compiler evaluate: needed.
  ].
^nil.
```

- getFacetValue: facet

Retourne le contenu de la facette facet en paramètre. Dans le cas où la facette demandée n'existe pas, une erreur est affichée :

```
getFacetValue: facet
  | val |
  ((facet = '$if-added') | (facet = '$if-needed') | (facet = '$value'))
  ifTrue: [
    val := (self facets at: facet ifAbsent: [ ^'' ]).
    ^val.
  ]
  ifFalse: [
    Dialog warn: ('Facette inconnue : ', facet printString) withCRs
  ].
  ^''
```

- getValue

La méthode getValue récupère la valeur du slot courant. Le fonctionnement de cette méthode a été expliqué dans la partie précédente, au point concernant le frame-context. La voici :

```
getValue
  | val slotName allSuperFrames ifNeeded |
  val := self getFacetValue: '$value'.
  ((val = '') not) & ((val = nil) not)
  "si $value existe"
  ifTrue: [ ^val ]
  ifFalse: [
    "sinon, essayer ifneeded $if-needed"
    ifNeeded := self executelfNeeded.
    (ifNeeded = nil)
    ifFalse: [ ^ifNeeded. ]
    ifTrue: [
      "sinon regarde ako"
      "cherche la 1ere superframe possedant un slot de meme nom"
      slotName := FrameCtxt S.
      allSuperFrames := self frame getAllSuperFrames.
      allSuperFrames do:
        [:f |
          (f hasSlot: slotName)
          ifTrue: [ ^(f slot: slotName) getValue. ]
        ]
    ]
  ]. ^nil.
```

- putFacetValue: facet value: val

Cette méthode permet de renseigner la facette passée en premier paramètre avec la valeur fournie par le deuxième paramètre. Si l'ajout s'est fait dans la facette \$value, alors on déclenche la procédure présente dans la facette \$if-added. Si la facette est inconnue, on affiche une erreur.

```
putFacetValue: facet value: val
  ((facet = '$if-added') | (facet = '$if-needed') | (facet = '$value'))
  ifTrue: [
    self facets at: facet put: val.
    "execute ifadded"
    (facet = '$value') ifTrue: [
      self executelfAdded.
    ].
  ]
  ifFalse: [ Dialog warn: ('Facette inconnue : ', facet printString) withCRs ].
```

CONCLUSION

Suite à l'écriture de ce programme, nous sommes très satisfait du résultat. En effet, notre solution est simple à utiliser et permet une grande souplesse dans son utilisation.

Pour une utilisation « en production », il y aurait quelques petites optimisations possibles, comme par exemple pouvoir trier les Frames et les slots selon plusieurs critères. On pourrait aussi imaginer un nombre de facettes non limités ainsi que de nouveaux attachement procéduraux, lors de la suppression d'un slot, par exemple.