

SYSTEME EXPERT

Albéric Martel - Romain Bouleis

TABLE DES MATIÈRES

Table des matières.....	2
Présentation des systèmes experts.....	3
La base de faits.....	3
La base de règles	3
Le moteur d'inférence.....	3
Domaine d'application.....	5
Base de faits.....	6
Classe Fait.....	6
Classe FaitsCollection.....	7
Base de règles.....	10
Classe Regle.....	10
Classe ReglesCollection.....	10
Moteur d'inférence.....	13
Classe Moteur.....	13
Exemple sur le domaine d'application.....	14

PRÉSENTATION DES SYSTÈMES EXPERTS

Un système expert est un outil permettant de mettre en application un type d'intelligence artificielle. Son rôle est de reproduire le raisonnement d'un expert dans un domaine donné. Le premier système expert se nomme Dendral et permet d'identifier des constituants chimiques.

Pour réaliser une expertise, il doit proposer des solutions à des problèmes qui lui sont posés. Il fait cela en posant des questions aux utilisateurs et en fonction de leurs réponses, pose de nouvelles questions tout en restant cohérent avec les précédentes jusqu'à ce qu'il n'y ait plus de questions cohérentes à poser et qu'il ait proposé une solution. Par exemple, dans le système expert nommé Mycin permettant d'identifier des agents infectieux, une question peut porter sur le type de l'organisme, et propose deux choix, il est un bâtonnet ou une coque. Si la réponse donnée est bâtonnet, une des questions restant possibles ne pourra pas porter sur des médicaments permettant d'éradiquer les coques.

Pour fonctionner, un système expert est représenté par les éléments suivants: une base de connaissances constituée d'une base de faits et d'une base de règles puis un moteur d'inférence.

LA BASE DE FAITS

Elle contient les différents faits utiles à notre application. Les faits sont des variables décrivant le monde. Ils sont représentés par un nom et un état. On les utilise pour conditionner l'exécution des règles par le moteur d'inférence.

LA BASE DE RÈGLES

La base de règles contient les connaissances expertes, c'est-à-dire qu'elles représentent les raisonnements effectués par un expert. Elles sont appelées les unes à la suite des autres afin de créer des enchaînements de raisonnements. Tous ces raisonnements peuvent être représentés sous la forme de règles de production du type « Si *condition vraie* alors exécuter *action* ». Toutefois, cette représentation peut varier suivant le contexte de l'application.

LE MOTEUR D'INFÉRENCE

Un moteur d'inférence permet de conduire des raisonnements logiques en utilisant conjointement la base de faits et la base de règles. Selon différentes stratégies, le moteur d'inférence utilise des règles, les interprète, les enchaîne jusqu'à arriver à un état représentant une condition d'arrêt. Ces dernières dépendent du moteur et de la base de connaissances implémentée. En général, l'exécution de règles par le moteur d'inférence influe sur l'état des faits et éventuellement sur les autres règles.

Un moteur d'inférence peut exécuter des règles suivants différentes méthodes d'invocation:

1. Chaînage avant

Un moteur d'inférence fonctionne dans ce mode lorsque les faits de la base de faits représentent des informations dont la vérité a été prouvée. C'est-à-dire que ce mode de fonctionnement va des données (les faits) vers les buts.

2. Chaînage arrière

Un moteur d'inférence fonctionne dans ce mode lorsqu'il part d'un fait que l'on souhaite établir, qu'il recherche toutes les règles qui concluent sur ce fait, qu'il établit la liste des faits qu'il suffit de prouver pour qu'elles puissent se déclencher puis qu'il applique récursivement le même mécanisme aux autres faits contenus dans cette liste.

3. Chaînage mixte, chaînage bidirectionnel

Le chaînage mixte utilise les deux chaînages présentés ci-dessus. On peut donc partir des données pour aller vers les buts et également d'un fait inconnu pour aller vers les données permettant de le retrouver.

Un système expert peut fonctionner de deux manières différentes :

1. Système monotone

Dans un système monotone, à chaque cycle du moteur d'inférence, un groupe de règle peut être activé et leur exécution ne peut pas désactiver d'autres règles.

2. Système non monotone

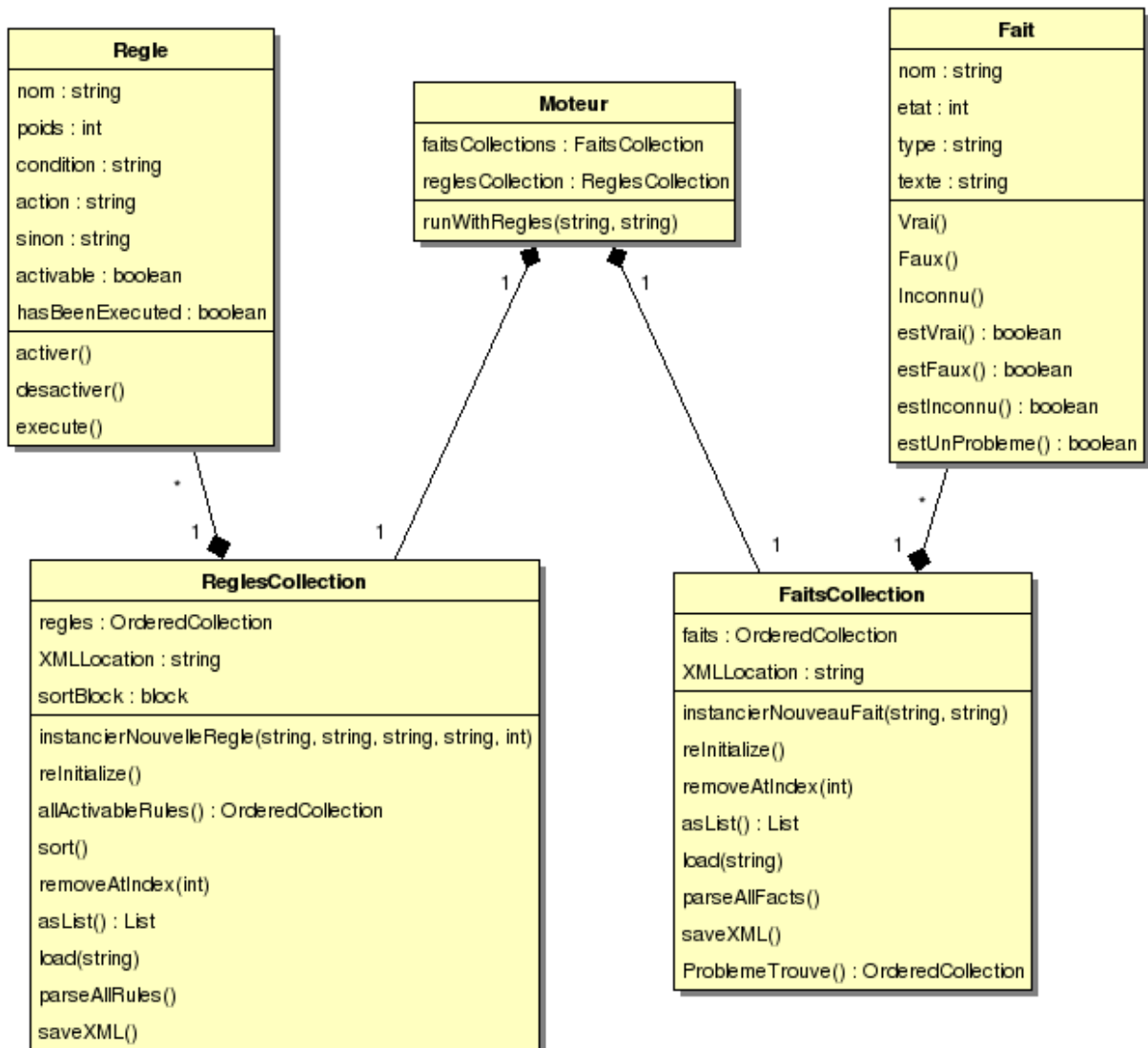
Dans un système non monotone, le moteur d'inférence choisit d'activer une seule règle à chaque cycle, éventuellement en fonction d'un poids précédemment attribué à cette règle. Son activation peut modifier l'état de la base de faits et donc l'activabilité de certaines règles.

DOMAINE D'APPLICATION

Pour notre application, nous avons choisi de développer une base de faits et une base de règles dont le but est de résoudre des problèmes de connexion à internet. Toutefois, notre système est assez générique pour permettre de résoudre des problèmes d'autre types. Il suffit donc de créer une base de fait et une base de règles spécifiques à un autre domaine et cela fonctionnera.

L'architecture de notre application est représentée dans le diagramme de classes ci-dessous. La base de faits est représentée par la classe FaitsCollection, la base de règles par la classe ReglesCollection et le moteur d'inférence par la classe Moteur. Toutes ces classes représentent le modèle. Nous avons 5 interfaces liées à ce modèle :

- un menu général.
- un éditeur de règles lié à ReglesCollection listant les règles et les informations liées.
- un éditeur d'une règle lié à la classe Règle permettant de modifier les informations d'une règle.
- Deux éditeurs de faits fonctionnant sur le même principe que les éditeur de règles.



BASE DE FAITS

Notre base de faits est bâtie sur un fichier XML. Ce fichier XML est constitué d'un élément racine nommé "facts" qui peut comporter plusieurs fois l'élément "fact" représentant chacun un fait. Cet élément "fact" est pour sa part composé de 2 autres éléments qui sont "nom" représentant le nom du fait et "texte" qui représente la description du fait.

Les faits peuvent être de deux types différents:

- Un fait devant être évalué
- La solution au problème posé par l'utilisateur. Pour ce type de fait, le nom commence par PROBLEME_.

Le code ci-dessous est tiré des classes Fait et FaitsCollection.

CLASSE FAIT

```
Smalltalk defineClass: #Fait
superclass: #{Core.Object}
indexedType: #none
private: false
instanceVariableNames: 'etat type nom texte '
classInstanceVariableNames: ''
imports: ''
category: 'TP - Systemes Experts'
```

- initialize

```
self Inconnu.
^self
```

- Faux

```
etat := 0.
```

- Inconnu

```
etat := -1.
```

- Vrai

```
etat := 1.
```

- printEtat

```
(etat = -1) ifTrue: [^'Inconnu'].
(etat = 0) ifTrue: [^'Faux'].
(etat =1) ifTrue: [^'Vrai'].
^'?'.
```

- estFaux

```
"si le fait est inconnu"
(self estInconnu & self estUnProbleme not) ifTrue: [
    (Dialog confirm: texte withCRs)
        ifTrue: [etat := 0] ;
        ifFalse: [etat := 1].
].
^(etat = 0)
```

- estInconnu

```
^(etat = -1).
```

- estUnProbleme

```
^ (type = 'PROBLEME').
```

- estVrai

```
"si le fait est inconnu"  
(self estInconnu & self estUnProbleme not) ifTrue: [  
    (Dialog confirm: texte withCRs)  
        ifTrue: [etat := 1] ;  
        ifFalse: [etat := 0].  
].  
^(etat = 1)
```

CLASSE FAITS COLLECTION

```
Smalltalk defineClass: #FaitsCollection  
superclass: #{Core.Object}  
indexedType: #none  
private: false  
instanceVariableNames: 'faits XMLLocation '  
classInstanceVariableNames: ''  
imports: ''  
category: 'TP - Systemes Experts'
```

- asList

```
^faits asList
```

- problemeTrouve

```
^(faits select: [ :f | (f estUnProbleme) ]) select: [ :f | f estVrai ].
```

- reInitialize

```
faits do: [ :f | f Inconnu.].
```

- removeAtIndex: anIndex

```
faits removeAtIndex: anIndex.
```

- instancierNouveauFait: nom details: texte

```
| createVariable createInstance type pos |  
  
type := ''.  
pos := (nom indexOf: $_)-1.  
pos > 1 ifTrue: [ type := nom copyFrom: 1 to: pos. ].  
  
createVariable :=  
    'Smalltalk defineSharedVariable: #',nom,  
    ' private: false',  
    ' constant: false',  
    ' category: ''TP - Systemes Experts'',  
    ' initializer: ''nil'''.  
  
createInstance := nom,':= Fait new type:','', type ,'' ; nom:','',nom, '' ;  
texte:', '', texte, '''.  
  
Compiler evaluate: createVariable.  
faits add: (Compiler evaluate: createInstance).
```

- load: aFile

```
| fileName |
fileName := aFile asFilename asLogicalFileSpecification.
fileName exists
  ifTrue: [
    XMLLocation := fileName.
    ^self parseAllFacts.
    "^ 'File ok'."
  ]
  ifFalse: [ ^ 'File does not exists'. ].
```

- parseAllFacts

```
| parser xmlDocument root factsCollection nom texte nomElement texteElement |
faits := OrderedCollection new.

"charge le parser xml"
parser := XML.XMLParser new.

"parse le fichier"
parser validate: false.
xmlDocument := parser parse: XMLLocation.

"recupere la racine"
root := xmlDocument root.

"recupere tous les faits"
factsCollection := (root elementsNamed: 'fact') select: [ :el | el isElement ].

"initialise les faits"
faitsCollection do: [:f |
  nom := 'UNDEFINED'.
  texte := ' '.

  nomElement := (f elementNamed: 'nom') elements.
  texteElement := (f elementNamed: 'texte') elements.

  (nomElement size > 0) ifTrue: [nom := (nomElement at: 1) characterData ].
  (texteElement size > 0) ifTrue: [texte := (texteElement at: 1)
characterData ].
  self instancierNouveauFait: nom details: texte.
].
```

- saveXML

```
|str newDoc nom etat elements fact writer |
newDoc := XML.Document new.
newDoc addNode: (XML.PI name: 'xml'
text: 'version="1.0" encoding="UTF-8" ').
newDoc addNode: (XML.Element tag: 'facts').

faits do: [:it |
  nom := XML.Element tag: 'nom'.
  nom addNode: (XML.Text text: it nom).
  etat := XML.Element tag: 'texte'.
  etat addNode: (XML.Text text: it texte).

  elements := OrderedCollection new.
  elements add: nom;add: etat.
  fact := XML.Element tag: 'fact' elements: elements.
  newDoc root addNode: fact.
].
```

```
str := XMLLocation asFilename writeStream.  
writer := XML.SAXWriter new output: str.  
[newDoc saxDo: writer] ensure: [str close].
```

BASE DE RÈGLES

Tout comme la base de faits, la base de règles se présente sous la forme d'un fichier XML avec comme élément racine "regles". A l'intérieur de cet élément, on trouve donc chaque règle. Une règle est représentée par l'élément "regle" et l'attribut "poids" et nécessite les éléments suivants:

- "nom" : c'est le nom de la règle.
- "si" : c'est la condition de la règle.
- "alors" : si la condition est satisfaite alors cette partie sera exécutée.
- "sinon" : dans le cas où la condition n'est pas satisfaite, c'est cette partie qui sera exécutée.

Les actions possibles sont le changement d'état des faits ainsi que la possibilité d'activer ou désactiver d'autres règles.

Par défaut, une règle sera considérée comme activable.

CLASSE REGLE

```
Smalltalk defineClass: #Regle
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'poids nom condition action sinon activable
hasBeenExecuted '
  classInstanceVariableNames: ''
  imports: ''
  category: 'TP - Systemes Experts'
```

- initialize

```
activable := true.
hasBeenExecuted := false.
^self
```

- activer

```
hasBeenExecuted = false ifTrue: [activable := true].
```

- desactiver

```
activable := false.
```

- execute

```
"execute la regle"
(Compiler evaluate: condition)
  ifTrue: [Compiler evaluate: action.] ;
  ifFalse: [Compiler evaluate: sinon.].

"desactive la regle"
activable := false.
hasBeenExecuted := true.
```

CLASSE REGLES COLLECTION

```
Smalltalk defineClass: #ReglesCollection
  superclass: #{Core.Object}
  indexedType: #none
```

```
private: false
instanceVariableNames: 'regles XMLLocation sortBlock '
classInstanceVariableNames: ''
imports: ''
category: 'TP - Systemes Experts'
```

- initialize

```
sortBlock := [:a :b | a poids >= b poids].
^self
```

- allActivableRules

```
^regles select: [:r | r activable.].
```

- asList

```
self sort.
^ regles asList.
```

- instancierNouvelleRegle: aName si: aCondition alors: anAction sinon: anotherAction poids: aPoids

```
| createVariable createInstance |
createVariable :=
  'Smalltalk defineSharedVariable: #',aName,
  ' private: false',
  ' constant: false',
  ' category: ''TP - Systemes Experts'',
  ' initializer: ''nil'''.
createInstance :=
  aName,':= Regle new
  nom:',aName, ' ;
  condition:',aCondition, ' ;
  action:',anAction, ' ;
  sinon:',anotherAction, ' ;
  poids:',aPoids, ' '.
Compiler evaluate: createVariable.
regles add: (Compiler evaluate: createInstance).
```

- reInitialize

```
regles do: [:r |
  r initialize.
].
regles sort: sortBlock.
```

- removeAtIndex: aIndex

```
regles removeAtIndex: aIndex.
```

- sort

```
regles sort: sortBlock.
```

- load: aFile

```
| fileName |
fileName := aFile asFilename asLogicalFileSpecification.
fileName exists
  ifTrue: [
```

```

XMLLocation := fileName.
^self parseAllRules.
"^ 'File ok'."
]
iffalse: [ ^ 'File does not exist'.].

```

- parseAllRules

```

| parser xmlDocument root reglesCollection nom condition action sinon poids |
regles := OrderedCollection new.

"charge le parser xml"
parser := XML.XMLParser new.

"parse le fichier"
parser validate: false.
xmlDocument := parser parse: XMLLocation.

"recupere la racine"
root := xmlDocument root.

"recupere toutes les regles"
reglesCollection := (root elementsNamed: 'regle') select: [ :el | el isElement ].

"instancie les regles"
reglesCollection do: [ :r |
    nom := ((r elementNamed: 'nom') elements at: 1) characterData.
    condition := ((r elementNamed: 'si') elements at: 1) characterData.
    action := ((r elementNamed: 'alors') elements at: 1) characterData.
    sinon := ((r elementNamed: 'sinon') elements at: 1) characterData.
    poids := (((r attributes) detect: [ :attr | attr tag type = 'poids' ])
value).

    self instancierNouvelleRegle: nom si: condition alors: action sinon: sinon
poids: poids.
].

```

- saveXML

```

|str newDoc regle nom condition action sinon elements attributes writer |
newDoc := XML.Document new.
newDoc addNode: (XML.PI name: 'xml' text: 'version="1.0" encoding="UTF-8" ').
newDoc addNode: (XML.Element tag: 'regles').

regles do: [:it |
    nom := XML.Element tag: 'nom'.
    nom addNode: (XML.Text text: it nom).
    condition := XML.Element tag: 'si'.
    condition addNode: (XML.Text text: it condition).
    action := XML.Element tag: 'alors'.
    action addNode: (XML.Text text: it action).
    sinon := XML.Element tag: 'sinon'.
    sinon addNode: (XML.Text text: it sinon).

    elements := OrderedCollection new.
    elements add: nom;add: condition;add: action;add: sinon.
    attributes := OrderedCollection new.
    attributes add: (XML.Attribute name: 'poids' value: it poids printString).
    regle := XML.Element tag: 'regle' attributes: attributes elements: elements.
    newDoc root addNode: regle.
].

str := XMLLocation asFilename writeStream.
writer := XML.SAXWriter new output: str.
[newDoc saxDo: writer] ensure: [str close].

```

MOTEUR D'INFÉRENCE

Nous avons choisi de développer un moteur d'inférence par chaînage avant et non monotone.

Tout d'abord le moteur récupère toutes les règles activables. Ensuite, tant qu'aucun fait représentant une solution (dont le nom commence par PROBLEME_) n'est dans l'état vrai et qu'il existe encore des règles activables, le moteur parcourt les règles de la liste précédemment créée en commençant par la règle de plus fort poids. Après exécution d'une règle, la règle avec le poids immédiatement inférieur est exécutée.

Lors de l'exécution d'une règle, si le fait sur lequel porte la question est à l'état « Inconnu », une question est posée à l'utilisateur. Cette question est récupérée dans l'élément «texte» de l'état sur lequel porte la condition de la règle. L'utilisateur répond par «oui» ou par «non», ce qui peut, suivant la réponse de l'utilisateur, activer ou désactiver d'autres règles et modifier l'état de faits. Si un fait commençant par «PROBLEME_» devient vrai alors le moteur s'arrête et affiche le texte du problème, qui dans ce cas correspond à sa solution.

CLASSE MOTEUR

```
Smalltalk defineClass: #Moteur
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'reglesCollection faitsCollection '
  classInstanceVariableNames: ''
  imports: ''
  category: 'TP - Systemes Experts'
```

- runWithRegles: regles withFaits: faits

```
| allActivablesRules |
faits reInitialize.
regles reInitialize.

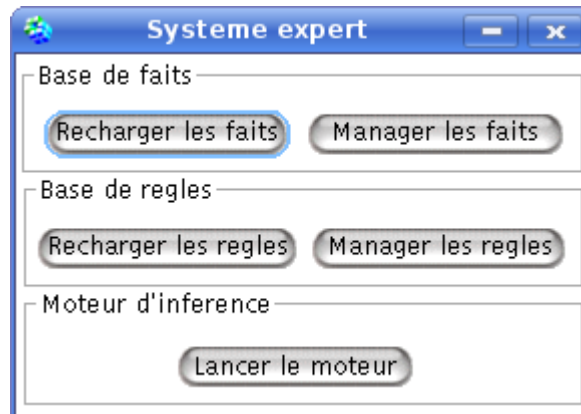
allActivablesRules := regles allActivableRules.

[((faits problemeTrouve size < 1) & (allActivablesRules size > 0)] whileTrue: [
  (allActivablesRules at: 1) execute.
  allActivablesRules := regles allActivableRules.
  faits changed.
].

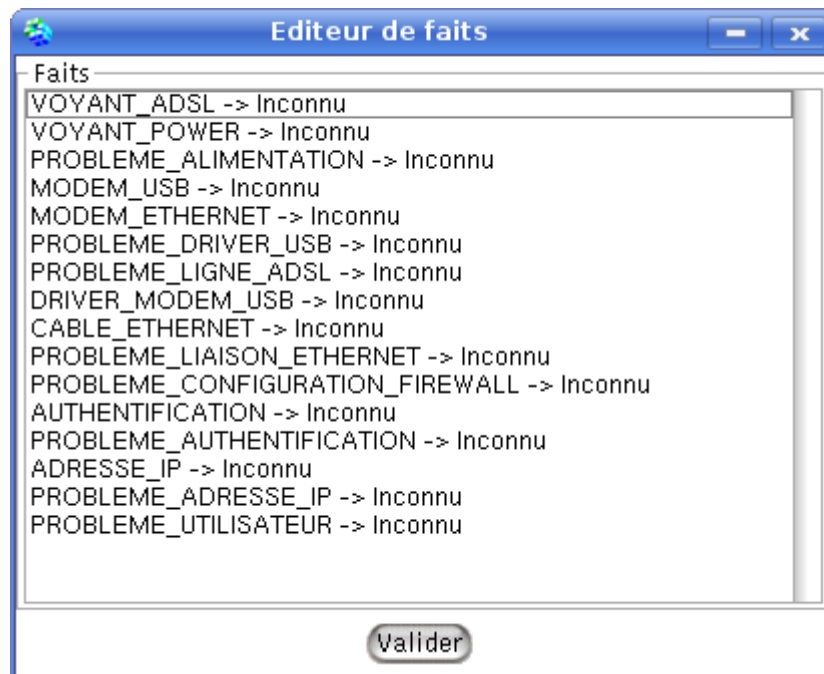
(faits problemeTrouve size > 0) ifTrue: [
  "Transcript show: (faits problemeTrouve at: 1)."
  Dialog warn: (faits problemeTrouve at: 1) texte withCRs.
].
```

EXEMPLE SUR LE DOMAINE D'APPLICATION

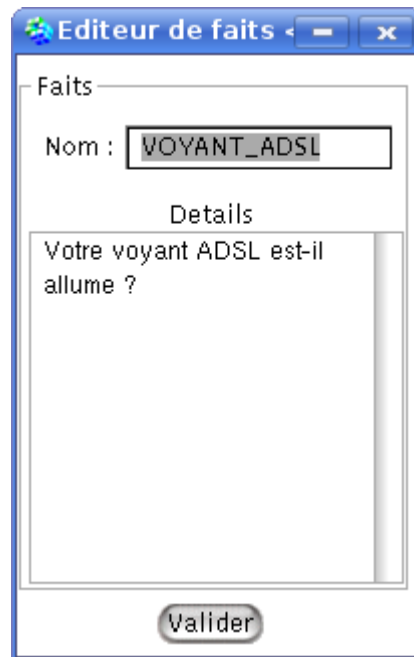
- Menu



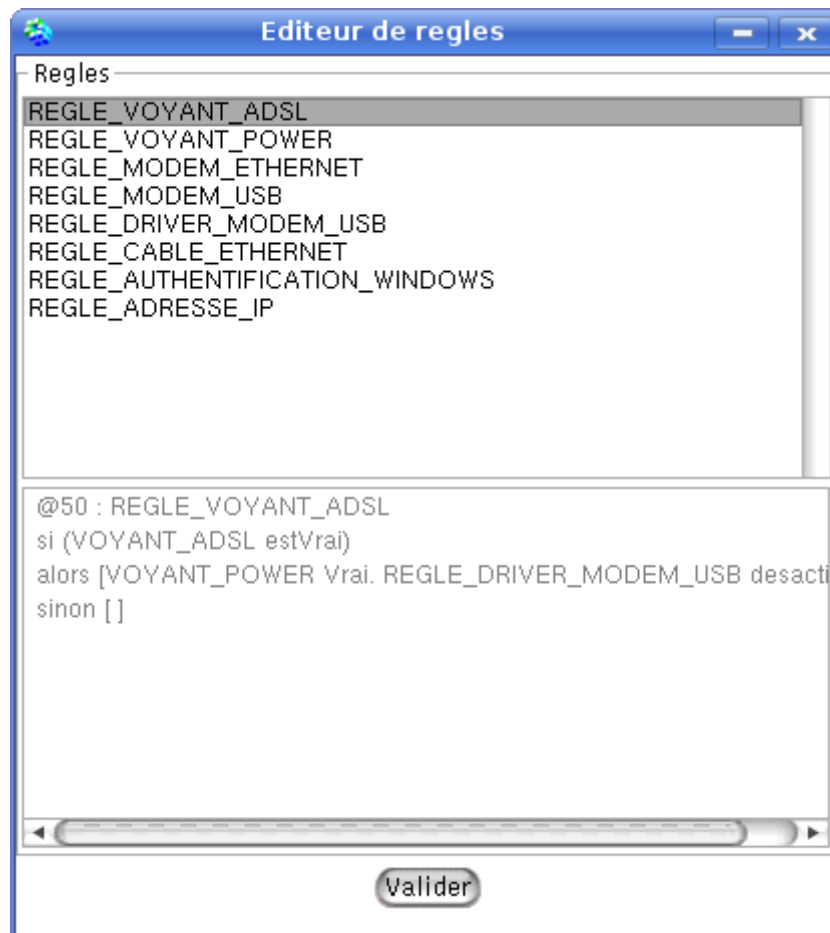
- Gestionnaire de faits



- Éditeur de faits



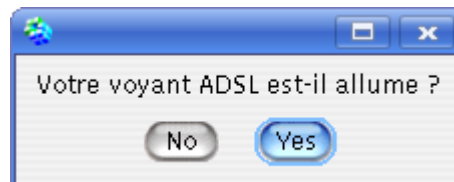
- Gestionnaire de règles



- Éditeur de règles



- Exécutions d'une règle



- Résolution du problème

